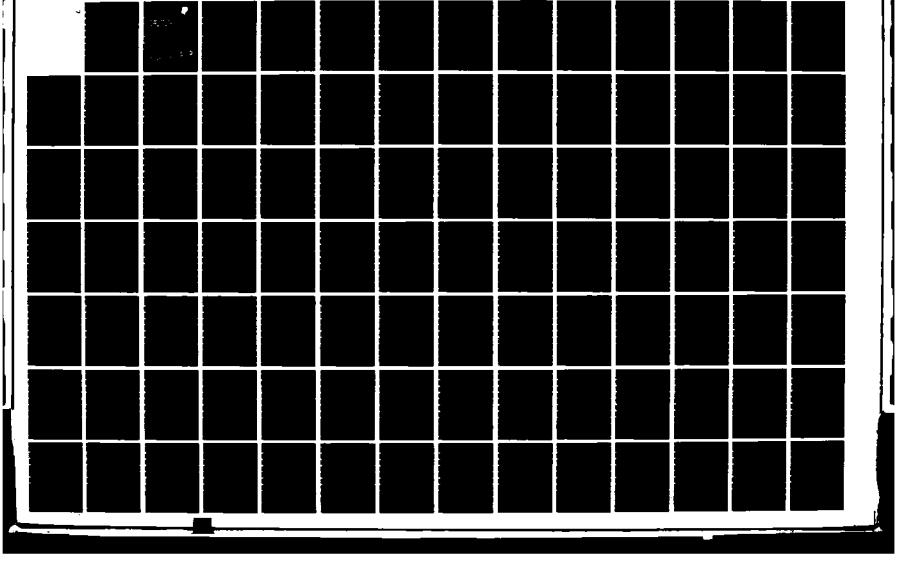
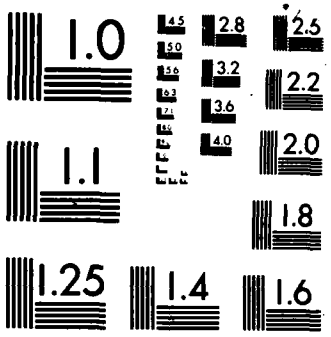
AD-A141 500 SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS..(U) GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUT.. P H ENSLOW ET AL. 1/2

UNCLASSIFIED JAN 84 GIT-ICS-82/16-VOL-1 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12)

RADC-TR-83-238, Vol I(of two)
Final Technical Report
January 1984

AD-A141 500

SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS

Georgia Institute of Technology

Philip H. Enslow, Jr.; N. J. Livesey; Richard J. LeBlanc
and Martin S. McKendry

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC FILE COPY

DTIC
ELECTE
MAY 2 3 1984
S D
E

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441

84 05 22 006

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-83-238, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: Thomas F. Lawrence

THOMAS F. LAWRENCE
Project Engineer

APPROVED:

JOHN J. MARCINIAK, Colonel, USAF
Chief, Command and Control Division

FOR THE COMMANDER: John A. Ritz

JOHN A. RITZ
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (COTD) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document requires that it be returned.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-83-238, Vol I (of two)	2. GOVT ACCESSION NO. AD-A142 500	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 20 Aug 81 - 31 Dec 82
		6. PERFORMING ORG. REPORT NUMBER GIT-ICS-82/16
7. AUTHOR(s) Philip H. Enslow, Jr. Richard J. LeBlanc N. J. Livesey Martin S. McKendry		8. CONTRACT OR GRANT NUMBER(s) F30602-81-C-0249
9. PERFORMING ORGANIZATION NAME AND ADDRESS Georgia Institute of Technology School of Information and Computer Science Atlanta GA 30332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 31011G R2440101
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (COTD) Griffiss AFB NY 13441		12. REPORT DATE January 1984
		13. NUMBER OF PAGES 150
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Same		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Thomas F. Lawrence (COTD)

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Distributed System Support Capabilities
Fully Distributed/Loosely Coupled Processing Systems
Software Development Tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The development and operation of very loosely-coupled distributed processing systems presents several new challenges. These "challenges", or differences from the techniques applicable to centralized systems, result primarily from the environment that is involved --- a multiplicity of logical and physical resources that are very loosely-coupled, a highly distributed and decentralized control system, and the autonomous and asynchronous operation of the various components. This report identifies

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the system support capabilities necessary to support the design, analysis, implementation, operation, utilization, and management of fully distributed, loosely-coupled, data processing systems. These system support capabilities are divided into three categories - software development support tools, distributed system design facilities, and operational support capabilities. Selected support capabilities are described, together with the rationale for the services that they will provide. Estimates are presented for the resources and facilities required to design and implement selected support capabilities. Also provided is a development priority for the support capabilities. The appendix to the report (Volume II) contains several papers providing in-depth discussions of various support capabilities and their features.

ACKNOWLEDGEMENTS

Support for the preparation of this report and most of the research on which it is based was provided by the U.S. Air Force, Rome Air Development Center under contract F30602-81-C-0249. In addition, general support for the Georgia Tech Research Program in Full Distributed Processing Systems has been provided by the Office of Naval Research under contract N00014-79-C-0873 as part of the ONR Selected Research Opportunities program. Dr. Enslow has also received support as a consultant in the area of software tools for embedded distributed systems under RADC contract F30602-81-C-0142, "Distributed Processing Tools Definition Study," with General Dynamics, Data Systems Division, Fort Worth, Texas.

Accession For
NTIS GRA&I
DTIC TAB
Unannounced
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

DTIC COPY INSPECTED

VOLUME 1

TABLE OF CONTENTS

VOLUME 1

LIST OF FIGURES

LIST OF TABLES

VOLUME 2

TABLE OF CONTENTS

VOLUME 2

LIST OF FIGURES

LIST OF TABLES

SECTION 1
INTRODUCTION

1.1 GENERAL

The topic of distributed systems is a major research area at the Georgia Institute of technology with the particular class of systems being examined, Fully Distributed Processing Systems --- FDPS, described in a definitional paper by Enslow [Ensl78]. In recent years, the phrase "distributed systems" has become an extremely popular term for both research and marketing; thus the meaning of the term has become very imprecise. For that reason, we at Georgia Tech have further identified our particular area of interest as "fully" distributed processing systems (FDPS). The major factor differentiating our work from that of others is that we are assuming a network of very loosely coupled processors. This is an important distinction to bear in mind, since our view of distributed systems is somewhat different from that of other researchers as a result of assuming this characteristic.

Conceptually, an FDPS consists of a loosely-coupled network of independent machines. Each machine is capable of communicating with other machines and controls a set of local physical and logical resources (e.g., processors, memory, files, devices, etc.). The machines are autonomous in that each processor or server has final responsibility for the control of the resources it provides. A layer of control is imposed on this network of machines to achieve unification of resources, cooperation, and system transparency. It is assumed that all machines, while retaining their autonomy, follow a common master plan to attain effective cooperation between the loosely-coupled logical as well as physical resources.

The primary goal of the Georgia Tech Research Program in Fully Distributed Processing Systems is to develop the technology necessary to design, implement and operate very loosely-coupled systems. Such systems should be capable of operating in dynamic system configurations with a high degree of cooperation in providing services requested from the system as a whole. The various research issues that have been identified thus far include such topics as distributed operating systems, programming languages, theoretical and formal studies, distributed data bases, physical interconnection and message transportation, fault tolerance, and security, among others.

This particular report discusses system support capabilities (SSC's) to support the activities of design, analysis, implementation, utilization, and management control of fully-distributed/loosely-coupled processing systems.

1.2 PURPOSE OF THIS STUDY

It is accepted that there will be some data processing applications or collections of applications for which some form of distributed processing is the only reasonable design philosophy. This study does not address the basis, rationale, or benefits and costs of such a decision. It is assumed that the decision to distribute has already been made. It is recognized that the basis for making such a decision has not yet been adequately studied. Some work on that topic is being performed under the same major research program which also included this particular project --- the Georgia Institute of Technology Research Program in Fully Distributed Processing System. That work will not be reported on here. However, earlier as well as other current work in the FDPS Research Program has presented persuasive arguments as to the requirement for very loose coupling, both logical and physical, in large-scale distributed systems. In fact, extremely loose-coupling is one of the fundamental system concepts of Fully Distributed Processing Systems; and that feature is accepted as a basic characteristic of the distributed systems considered as the target for the work performed under this immediate study.

Loosely-coupled distributed systems will pass through the same life-cycle phases as centralized systems and will require many of the same support capabilities as such systems. However, the exact nature of loosely-coupled distributed systems presents additional requirements for new support capabilities as well as changes or extensions to the support that would be provided for the analysis, design, implementation, and operation of centralized systems. The scope of this study covers new capabilities as well as extensions to "existing" ones.

1.2.1 Extensive Support Capabilities Are Essential

There are three major activities to be supported by the "capabilities" being considered in this report

- Designing a specific distributed system
- Producing the software to implement that system
- Operating that system

All three of these activities are greatly complicated by the basic characteristics of "distribution" considered in its broadest sense.

- The presence of multiple execution environments, operating simultaneously with almost total asynchrony and often non-homogeneity, creates perhaps the greatest problems in all three activities.

- The problems facing software development are the same as those found in centralized uni-processor systems with the added complications of having multiple target environments as well as multiple development environments.

The only solution that appears viable is a well-developed and extensive set of software and hardware support capabilities.

1.2.2 Scope and Outline of This Project

The original scope of work under this project consisted of three major steps.

Step 1. Investigate the need for system support capabilities

a. Identify capabilities required to support design, analysis, implementation, utilization, and management control of fully distributed loosely-coupled data processing systems.

b. Document the control problem associated with each activity.

c. Identify specific system support capabilities required to support each activity.

d. Categorize the system support capabilities identifed into the following two categories

I - Most essential, urgently required

II - Secondary importance

and estimate resources and facilities required to implement each capability.

Step 2. Implement and demonstrate those "essential" (i.e., Category I) support capabilities selected/specified by the government.

Step 3. Implement and demonstrate those "secondary" (i.e., Category II) support capabilities selected/specified by the government.

Present plans do not contemplate the execution of Steps 2 and 3 at this time, and the work plan has been modified accordingly. The primary activity is now just the first step, the identification of systems support capabilities required. The remaining resources will be utilized to perform the preliminary design of some of the most essential capabilities.

1.3 THE LIFE CYCLE OF DISTRIBUTED SYSTEMS

One important aspect to consider in examining the requirements for support capabilities is the specific environmment in which a given support capability is to be employed. For the purpose of this study, the application environment will be identified by reference to a phase or a set of phases in the overall life cycle of a distributed system.

In this study the various phases or activities of the life cycle are, in chronological order,

- Problem Analysis and Functional Design
- Logical System Design
- Program Implementation
- Unit Test
- System Integration
- System Test
- Program Distribution and Installation
- System Operation and Utilization
- System Maintenance

Just as with centralized systems, to which this list is equally applicable, there are a number of feedback paths present in the complete life cycle.

1.4 CATEGORIES OF SUPPORT CAPABILITIES AND THEIR APPLICATION

As this study has progressed, a large number of different system support capabilities applicable to the total life cycle of distributed systems have been identified. As the list expanded, it became obvious that a large amount of confusion was being created by the lack of a clear definition of the relationships between the various capabilities and their specific applicability. A major cause of this confusion was the absence of a clear distinction between the major categories of support capabilities. In addressing this particular problem, three major types of support capabilities have been identified:

- Software Development Support Tools
- System Design Support Facilities
- Operational Support Capabilities

1.4.1 Software Development Support Tools

The primary purposes of software development support tools are the production, maintenance, and management of the operational software systems, both operating systems and applications programs --- i.e., the production of software. Some confusion is caused by the word "software" in the title. It should be noted the "software" applies to the application of the tool or support capability, not the nature of the tool itself since nearly all of the support capabilities will be implemented in software, at least in part.

It is unfortunate that the designations "tool" and "support capability" have been widely used almost totally interchangeably. (We have been as guilty of this as anyone else.) However, using the terms in this manner was one of the major factors creating the confusion referred to above.

Because of the wide variety of support capabilities found within this single category, further subcategories are useful in examining the categorization and applicability of software support tools. The subcategories identified thus far are:

- Software Requirements/Specification Tools
- Software Design Tools
- Software Implementation (Programming) Tools
- Software Quality Assurance Tools
- Software Maintenance Tools
- Software Cross-Environment Tools
- Miscellaneous Software Utility Tools

- Software Management Tools

It should be noted that these subcategories are equally applicable to tools supporting centralized systems.

The list of subcategories given above will be utilized during this study when such subdivisions are required; however, that is not the only set that has been proposed. William Howden in discussing software development environments presents a five-way categorization [Howden].

- Requirements Tools and Methods
- Design Tools and Methods
- Coding Tools and Methods
- Verification Tools
- Management Tools and Techniques

Also, A.N. Haberman in [Riddle & Fairley] discusses his two-part classification

- Program Development Tools
- System Construction Tools

where examples of the first are the "classical tools such as compilers and editors" while the latter "emphasizes the importance of specifications and system version maintenance."

Another categorization methodology for software development tools has been proposed by the Software Tools Project of the Institute for Computer Sciences and Technology at the National Bureau of Standards. This methodology is based on a multi-dimensional taxonomy of tool features describing the characteristics of the input, the function, and the output of the tool. These three major features are further divided into two or three dimensions. In all there are 7 dimensions. In the list below the following notation is employed:

- **Basic processes of a tool**
  - Classes of tool features - Classification dimensions.
    - Specific tool features - multiple features in a single class may apply to a given tool.
- **Input**
  - Subject (i.e., Main input)
    - Text
    - VHLL (Very high level language)
    - Code
    - Data
  - Control Input
    - Commands
    - Parameters

- **Function**
  - Transformation (How is the subject manipulated)
    - Editing
    - Formatting
    - Instrumentation
    - Optimization
    - Restructuring
    - Translation
  - Static Analysis (Operations on the subject)
    - Auditing
    - Comparison
    - Complexity Measurement
    - Completeness Checking
    - Consistency Checking
    - Cost Estimation
    - Cross Reference
    - Data Flow Analysis
    - Error Checking
    - Interface Analysis
    - Management
    - Resource Estimation
    - Scanning
    - Scheduling
    - Statistical Anslysis
    - Structure Checking
    - Tracking
    - Type Analysis
    - Units Analysis
  - Dynamic Analysis (Operations during or after execution)
    - Assertion Checking
    - Constraint Evaluation
    - Coverage Analysis
    - Resource Utilization
    - Simulation
    - Symbolic Execution
    - Timing
    - Tracing
    - Tuning

- **Output**
  - User Output
    - Computational Results
    - Diagnostics
    - Graphics
    - Listings
    - Text
    - Tables
  - Machine Output
    - Data
    - Intermediate Code
    - Object Code
    - Prompts
    - Source Code
    - Text

Although this classification methodology was developed primarily to support, or force, complete descriptions of tools, it has also been useful in the context of this study to check for completeness of coverage in our consideration of the need for various software development tools.

1.4.1.1 Examples of Software Development Support Tools

Implementation tools such as compilers and editors are certainly the most common; however, there is beginning to be significant activity in the development of support capabilities in the other categories as well. In the initial edition of the "Software Engineering Automated Tools Index" published by Software Research Associates the breakdown was as follows:

Category	Number of Tools	Percentage of Total
Requirements/Specification Tools	20	3%
Design Tools	47	7%
Implementation Tools	210	32%
Quality Assurance Tools	132	20%
Maintenance Tools	119	18%
Project Management Tools	57	9%
Cross-Environment Tools	16	2%
Miscellaneous Utility Systems	40	6%
Research and Development Systems	7	1%

Examples of specific tools that fall in each subcategory are given below.

- **Requirement/Specification Tools**
  - Requirement/Specification Languages
  - Charts and Diagrams (both formal and informal)
    (e.g., HIPO, SADT, Dataflow, etc.)
  - Specification Cross-Reference Analyzer
  - Archiver/retriever for requirements specifications
- **Design Tools**
  - Formal Design Tools/Methodologies
    (e.g., PDL, Structured Design)
  - Automated Data Dictionary
  - Distributed Data Base and Transactions Processing Design Language
  - Module Interface Checker
  - Module Cross-Reference Analyzer
  - Automated Simulator Builder
  - Automated Archiver for Design Specifications

- **Implementation Tools**
  - Distributed Applications Programming Languages
  - Distributed Systems Implementation Languages
  - Editors
  - Text Managers (source and object code file systems)
  - Source Code Manager
  - Program Cross-Reference Analyzer
  - Language Processors
  - Compiler Development Tools
- **Quality Assurance Tools**
  - Flow Charter
  - Test Harnesses
  - Test Coverage Analyzer
  - Test Data Generator
  - Control Flow Analyzer
  - Data Flow Analyzer
- **Maintenance Tools**
  - Source Code Debugging
  - Trouble Report and Comment Tracking System
- **Cross-Environment Tools**
  - Cross Compilers
  - Environment Simulators
- **Miscellaneous Utility Tools**
  - Program Archiver
- **Management Tools**
  - Project Status Control
  - Project Status Report Generators
  - Build Plan Recorders
  - Configuration Manager
  - Cost Estimator
  - Version Manager

1.4.1.2 Applicability of Software Support Tools

The applicability of the various subcategories of tools to the different phases in the overall life cycle is fairly obvious from the name of each subcategory.

- **Software Requirements/Specification Tools**
  - Problem Analysis and Functional Design
- **Sofware Design Tools**
  - Logical System Design
- **Software Implementation Tools**
  - Program Implementation
- **Software Quality Assurance Tools**
  - Unit Test
  - System Test
  - System Maintenance
- **Software Maintenance Tools**
  - System Maintenance

- **Cross-Environment Tools**
  Program Distribution and Installation
- **Miscellaneous Software Utility Tools**
  System Maintenance
- **Software Management Tools**
  System Integration
  Program Distribution and Installation

1.4.2 System Design Support Facilities

"System Design Support Facilities" describe that collection of hardware and software facilities utilized to support the analysis, design, testing, experimentation, and monitoring of distributed processing systems. System design support facilities provide information about the distributed system, they do not directly produce or modify the software defining the operation of the system, nor do they directly support the operation of the system.

1.4.2.1 Examples of Hardware/Software Support Facilities

- **Capacity Requirement Estimators**
  Computation
  Storage
  Communication
- **Simulators**
  System
  Communications
  Transaction Processing
- **Load Emulators**
- **Monitors/Performance Measurement**
  Resource Utilization
  File Performance
- **Testbeds**
- **Redundancy Requirement Planner**
- **Fault-Tolerance Estimator**
- **Database Designers Workbench**

1.4.2.2 Applicability of System Design Support Facilities

System Design Support Facilities are applicable primarily to the analysis and design phases of the life cycle. The facilities are also useful to support testing and maintenance as well as management of operations.

1.4.3 Operational Support Capabilities

"Operational support capabilities" directly support the operation of the distributed processing system and are physically embedded in the operational software system. These capabilities provide those functions which are "unique" to distributed systems operations and are usually found in that por-

tion of the system software known as the "network operating system" (NOS) or "distributed operating system" (DOS). (In the logical model of system software as developed in the GIT FDPS Research Program, those functions performing tasks similar to those found in centralized systems are included in the "local operating system" (LOS). Although the LOS must interface with and interact with the NOS/DOS, the GIT model places all support of distribution in the NOS/DOS. It should be noted that current research at Georgia Tech indicates that it may be possible to effectively combine all these operations into a single global operating system.)

1.4.3.1 Examples of Operational Support Capabilities

- Access Control
- System Command Language
- Workload Distributor
- Resource Manager
- Task Graph Manager
- Interprocess Communication
- Scheduler
- Execution Manager
- File Manager
- Recovery Manager
- Communication Protocols

1.4.3.2 Applicability of Operating System Capabilities

Although operating system capabilities are primarily involved with the operation of the distributed system, several of them are also applicable to other phases of the life cycle as shown in the chart below in paragraph 1.5.

1.5 APPLICABILITY OF SYSTEM SUPPORT CAPABILITIES

The matrix shown in Chart 1 indicates the primary applicability of the various support capabilities identified in this study.

CHART 1

UTILIZATION OF SYSTEM SUPPORT CAPABILITIES DURING VARIOUS LIFE CYCLE ACTIVITES

	Anal	Desg	Impl	Unit Test	Sys Integ	Sys Test	Instl	Oper Util	Maint
SOFTWARE SUPPORT TOOLS									
Software Requirements/Specification Tools									
Rqmts/Spec Lang	X	X	X						X
Charts and Diagrams	X	X	X	X					X
Cross-Ref Analyzer	X		X						X
Archiver	X								X
Software Design Tools									
Form Design Tools	X	X			X				
Auto Data Dict		X							
DDB & Trans Proc		X							
Design Lang		X			X				
Interface Checker		X		X					X
Cross-Ref Analyzer		X							
Auto Simul Build		X							
Auto Archiver		X							
Software Implementation (Programming) Tools									
Dist Prog Lang			X						X
Dist Sys Impl Lang			X						X
Editors			X						X
Text Mgrs			X						X
Source Code Mgrs			X						X
Cross-Ref Analyzer			X						X
Compiler Development			X						X
Software Quality Assurance Tools									
Test Harness				X	X	X			X
Test Cover Anal				X	X	X			X
Test Data Gener				X	X	X			X
Control Flow Anal				X	X	X			X
Data Flow Anal			X	X	X	X			X
Software Maintenance Tools									
Source Code Debug				X	X	X			X
Report Tracking				X	X	X			X

Cross-Environment Tools									
Cross-Compilers			X				X		X
Environment Simuls				X	X	X			X
Miscellaneous Utility Tools									
Prog Archiver			X						X
Management Tools									
Proj Status Cont		X	X	X	X	X			X
Proj Status Reporter		X	X	X	X	X			X
Build Plan Recorder			X	X	X	X	X		X
Config Mgr			X		X				
Cost Est	X	X							
Version Mgr									X

	Anal	Desg	Impl	Unit Test	Sys Integ	Sys Test	Instl	Oper Util	Maint
DESIGN SUPPORT FACILITIES									
Resource Ests	X	X							
System Simulators	X	X							X
Comm Simulators	X	X				X			X
Trans Proc Simul	X	X							
Load Emuls						X			X
Monitors				X		X		X	X
Testbed				X		X			X
Redund Plan	X	X							X
Fault-Tol Est	X	X							X
DB Design WB	X	X							

OPERATIONAL SUPPORT CAPABILITIES									
Dist Access Control								X	
Dist Cmd Lang								X	
Workload Dist								X	
Resource Mgr								X	
Dist Task Graph Mgr								X	
Dist Scheduler								X	
Dist Execution Mgr								X	
Dist IPC								X	
Dist File Mgr								X	
Dist File Perf Analyzer	X					X		X	X
Dist Recov Mgr								X	
Dist Comm Protocols								X	

1.6 SUPPORT CAPABILITIES AND SYSTEM FUNCTIONALITY

There are a number of points of view or approaches that can be taken when studying the characteristics, interrelationships, and relative development priorities of support capabilities. The two analyses presented thus far focused on the nature or form of the support capability and on the life-cycle activity(ies) that each supports. Another very important point of view is analyzing what support capabilities are required or implied by the functionality to be provided by the operational system.

Distributed Resource Utilization

- Access Control
- Work Distribution and Resource Management
- Distributed Process Execution Manager
  - Distributed Task Graph Manager
- Distributed File System
  - Distributed Name Server
- Distributed Interprocess Communication

User Services

- Distributed Command Language
  - Distributed Software Tools
  - Distributed Compiling Shell
- Remote Access
- Mail

Communication System Support

- Protocols for Interprocess and Interprocessor Communication
  - Low-level
  - Session
  - Reliable Broadcast
- Network Simulators
- Communication Requirements Estimator

Fault-Tolerance Support

- Recovery/Rollback Manager
- Activity Journalling
- Redundancy Requirement Planner
- Fault-Tolerance Estimator

Distributed Data Base Operations

- Concurrency Control Mechanism
- Transaction Processing Manager
- Security Control Mechanism (Multi-Level)
- Application Design Tool
- Consistency Verifier
- Archive Support
- Transaction Processing Simulator

Life-Cycle System Control and Administration

Trouble/Change Report and Fix Tracking System
Code Version Control
Performance Measurement
Quality Metrics

1.7 OTHER WORK IN THIS AREA

"Distributed systems" has become an extremely popular topic for research. In fact, this topic has probably become the most popular research area in computing today. The large amount of work being performed should make it very easy to locate solutions to many, if not, most of the problems/requirements described in this report. However, such is not the case.

When one studies the work done under the general title of "distributed systems", there is a grave danger of misunderstanding or misinterpreting the exact nature of the target system involved. More specifically, it is usually difficult, if not impossible, to determine the specific characteristics of the system to which the results are applicable, especially with respect to the degree of distribution that applies. In fact, in many instances it is obvious that the researcher has not completely defined the specific class(es) of systems to which his work applies. This task then falls on the reader, and his analysis is then based on incomplete data and often erroneous assumptions. The net result of this is that it is usually difficult, and often impossible, to accurately determine the applicability of published results in this field.

For this reason we will not attempt to survey, much less catalog, other work done in this area on the topic of support capabilities for distributed systems. However, there are two activities that should be mentioned --- the distributed computing research program of the U.S. Army Ballistic Missile Defense Advanced Technology Center (BMDATC), Huntsville, Alabama, and the "Distributed Processing Tools Definition Study" performed by the Data Systems Division of General Dynamics for the U.S. Air Force, Rome Air Development Center (RADC/ISIE), Contract F30602-81-C-0142.

1.7.1 BMDATC-P

This project has been underway for several years with the definitional phase starting in 1975. There have been a large number of contracts in this program covering, to varying degrees, almost all of the areas identified in this report. The concept of an "Integrated Tool Set" is proposed to support the steps "requirements design" through "software design" and "implementation." This program has also developed and installed a multi-computer testbed at Huntsville.

The types of target systems considered originally were not quite as loosely-coupled as those addressed by Georgia Tech; however, the focus has changed some over the years. The point of contact is BMDATC-P.

1.7.2 General-Dynamics (RADC) Project

This was a three phase project:

I - Study of hardware/software technologies for embedded distributed processing systems (EDPS); supporting the identification of techniques, requirements, and impacts for EDPS lifecycle phases.

II - Survey of existing EDPS tool inventories; developing the EDPS life cycle requirements with no near-term tools.

III - Analysis of EDPS problem areas; resulting in a prioritized list of candidate technologies for R&D and estimates of the effort involved in each as well as its potential benefit.

A variety of different types or classes of distributed systems were considered; however, the level of detail in the discussion often does not permit their exact definition. Heavy emphasis was given to object-oriented models.

1.8 ORGANIZATION OF THIS REPORT

The goal of this project was to identify, classify, and prioritize system support capabilities (SSC) as they relate to FDPS/loosely-coupled Systems. The description and discussion of individual support capabilities is organized based on the nature or form of the SSC involved. The three sections of the report containing these descriptions are:

Section 2: Software Development Support Tools

Section 3: Distributed System Design Support Facilities

Section 4: Operational Support Capabilities

These sections contain discussion of specific support capabilities or tools which we believe should be implemented or specific areas which should be researched to provide the basis for a later implementation. Each discussion contains the following subsections:

1. Short description of the support capability/tool
2. Background (Why this support capability is required)
3. Problems (general and specific) to be Solved
4. Proposed solutions (or initial approaches for research)
5. Relationship to other FDPS work and SSC's
6. Resources and schedule

Because of the very nature of this document, certain SSC's are more completely defined than others. In most cases the need or the desire for a specific support capability is recognized before the exact means to provide it is determined. Thus, the level of each discussion varies with our current understanding of the problem and schemes to implement solutions. Additional work past the definition stage has been on several of the support capabilities. This work has been identified in the discussions in Sections 2, 3, and 4;and there are several references to detailed material included in the Appendix to this report which is published as a separate volume.

Section 5 concludes the report addressing the priorities for the development of specific SSC's or groups of them.

1.9 REFERENCES

[Godfrey80] Godfrey, M.D., et. atl. Machine Independent Organic Software Tools, 1980.

[Howden82] Howden, William E., "Contemporary Software Development Environments," CACM, Vol 25, No 5, May, 1982, pp 318-329.

[NBS81] NBS, "Software Development Tools: A Reference Guide to a Taxonomy of Tool Features," Center for Programming Science and Technology, Institute for Computer Science and Technology, National Bureau of Standards, Washington, D.C., February, 1981, 12 pp.

[Riddle80] Riddle, W.E. and R.E. Fairley, Software Development Tools, Springer-Verlag, Berlin, 1980, 277+viii pp. (Proceedings of a workshop on Software Development tools held at Pinegree Park, Colorado, May, 1979. Workshop emphasized pre-implementation phases of Software development.) Of particular interest:

Haberman, A.N., "Tools for Software System Construction," pp. 10-21.

Lesser, V.R. and J.C. Wildeden, "Issues in the Design of Tools for Distributed Software System Development," pp. 22-39.

Nassi, I., "A Critical Look at the Process of Tool Development: An Industrial Perspective," pp. 40-51. "Discussion," pp. 52-61.

SECTION 2
SOFTWARE DEVELOPMENT SUPPORT TOOLS

2.1 DESIGN LANGUAGES

2.1.1 Introduction

Among the potential uses of a fully distributed processing system is the execution of distributed programs. Such programs consist of a collection of relatively loosely-coupled modules which together perform a single logical task. Such programs are termed "distributed" programs because the individual modules may (or in some cases, must) be executed in parallel on separate nodes of the FDPS. The possible motivations for such parallel executions include, among others: (1) operating on large quantities of data at the nodes where they are stored or (2) taking advantage of inherent parallelism in a task being implemented. As an example of the latter of these two motivations, a distributed compiler has recently been implemented as part of the FDPS research project, in order to study the advantages of organizing a compiler using a pipeline structure and thus executing the normal phases of a compilation in parallel [MILL82].

The development of software for distributed systems may require some new techniques throughout the software life cycle. One problem which must be addressed is how the various parts of a distributed program can be described as a single, conceptually unified program. At the programming language level, few existing or proposed languages which include features for expressing parallelism or concurrency provide much help. Information about how the parts of programs written in these languages interact can only be obtained by detailed examination of the code for the individual parts. One project currently in progress within our research program is concerned with the development of a set of language features (called PRONET) which allow the description of the interactions among a group of "processes" by way of a "network" specification. These specifications can be interpreted as abstract descriptions of the communication behavior of the processes which make up distributed programs and thus provide at least some of the conceptual unification we desire. We believe that it may be useful to apply the concepts and perhaps even the notation of PRONET network specifications as the basis of a design language for distributed software.

2.1.2 Background

The need for a design phase, prior to the implementation phase, has long been recognized ([PARN72], [LISK72], and [WIRT71]). Benefits which accrue from such a phase include reliability of software, productivity of programmers and maintainability of software. The design phase consists basically of determining what services are required of the software and then deciding how the software is to provide these services [PETE81].

The function of a design language is two-fold [JENS78]. First, a design language should allow the software designer to specify his ideas in a form that will be understood by other people. This is especially important in cases where more than one person is working on a project. The design language becomes the medium in which the designers communicate. Second, the design language should be machine processable. Programs can then be written which analyze the design. In this way, inconsistencies in the design can be caught by the design tools and brought to the attention of the designers.

Much work has been done in developing methodologies and notations for representing program design. A few include PDL [CAIN75], HOS [HAMI76], SARAH [ESTR78], TOPD [SNOW78], and FLEX [SUTT81].

2.1.3 Problems

The model of computation proposed for a FDPS is that of a co-operating network of processes [LEBL81]. These processes are independent in the sense that no processes control the behavior of another directly. Instead, the processes communicate by passing messages to one another. These messages allow the programs to exchange data and to coordinate their behavior.

This model allows the designer of distributed software to take advantage of any parallelism available in the problem he is attempting to solve. Also, the goal of breaking the problem into smaller, relatively independent subunits fits in well with traditional concepts of software design, for example, information hiding [PARN72]. However, the model also provides new challenges for the software designer.

One is that the designer must explicitly design the process structure to take advantage of any parallelism in the problem. He must design so that independent actions are performed by separate modules. The designer can no longer think in terms of shared memory. Instead, the components of his design exchange data by message-passing. The designer is also responsible for synch-

ronizing the efforts of the various processes, at least at a conceptual level.

Clearly, the designer of distributed software could benefit from a design language which would allow him to easily specify the above. Unfortunately, most existing design languages and methodologies do not meet the above goals. For the most part, they assume a hierarchy of program control. That is, there is some single component in the design which is responsible for coordination of all the others. The tools that they provide for specifying the synchronization and communication of the components in the design are not adequate. Another concern of distributed software is reliability -- what happens to the program as a whole if there is a failure at one of the machines in the system. The design language should allow the designer to take this into account.

2.1.4 Proposed Solutions

Efforts made for support of distributed software have already appeared in the form of implementation languages ([BRIN78] and [HOAR78]). Here at Georgia Tech, implementation of a compiler for PRONET is almost complete. (PRONET is described in Appendix C.) The language consists of two parts: a process specification notation and a network specification notation. The subunits would be implemented using the former notation; the interactions between the subunits would be described using the latter notation. Effort in the direction of design languages, however, has been lacking.

Recent work on the use of Ada as a design language is also relevant to this problem. (See [BOOC81] and various reports in Ada Letters.) The package and tasking facilities in Ada provide the designer with the ability to express a program as abstract objects. Thus, he can hide information and specify the cooperative aspect of the subunits.

Other work done specifically for distributed software includes that described in [YAU81]. Here, the data and functional specifications are considered separately. The program is broken up into components which interact only through shared resources or messages. However, the methodology used in this approach requires that interactions across processor boundaries be identified. This aspect of the work does not go along with the general philosophy of an FDPS. That is, the user of a FDPS should be able to view the system as one unified system, and that the system itself should normally be responsible for where work occurs.

We propose to add the concept of an abstract description of the communication behavior of parallel processes to existing design language concepts. The existing design language would be used to specify the compositions of the subunits. Network specifications like those of PRONET would indicate how these subunits interact. In addition, since there is a compiler underway for PRONET, a processor for the design language could utilize much of the work done there.

2.1.5 Relationship to Other FDPS Work

The FDPS project consists of many interrelated projects. Work in this area includes the implementation of a network operating system, a distributed system test-bed, a distributed execution monitor, and a distributed debugger. These projects represent major efforts in software engineering. The assistance of a design language in developing the programs would be invaluable in producing reliable, comprehensible systems. Work on a design language and a methdology for its use can proceed independently of these other projects.

2.1.6 Resources and Schedule

The major tasks required for this project are to survey these existing design languages, to choose one which is most compatible with our concept of independent processes communicating by message passing, and to integrate the PRONET specification concepts with that design language. This integration will include adding some consideration of reliability in the presence of node failures. Such a concept again goes beyond those found in typical design languages.

To cover a 12 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	2
Junior Staff (6 m-m/year)	6
Programmer (0 m-m/year)	0
Secretarial Support (1 m-m/year)	1

Equipment

Computer Time — moderate usage (for designing syntax of new design language)

Timing

First period of 3 months:
Examination of existing design languages

Last period of 9 months:
Extension of a design language to include new concepts described above

2.1.7 References

[BOOC81] G. Booch, "Describing Software Design in Ada," Sigplan Notices, Vol. 16, No. 9, p. 42-47, September, 1981.

[BRIN78] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept," Communications of the ACM, Vol. 21, NO. 11, p. 934-941, November, 1978.

[CAIN78] S. H. Caine and K. E. Gordon, "PDL -- A Tool for Software Design," Proceedings of the National AFIPS Computer Conference, p. 271-276, 1975.

[ESTR78] G. Estrin and I. A. Campos, "Concurrent Software System Design Supported by SARA at the Age of One," Proceedings of the 3rd Conference on Software Engineering, p. 230-242, 1978.

[HAMI76] M. Hamilton and S. Zeldon, "Higher Order Software -- A Methodology for Defining Software," IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, p. 9-32, January, 1976.

[HOAR78] C. A. R. Hoare, "Communicating Sequential Processes," Communication of the ACM, Vol. 21, No. 8, p. 666-677, August, 1978.

[JENS78] R. W. Jensen and C. C. Tonies, Software Engineering, Prentice-Hill, Englewood Cliffs, N. J., 1978.

[LEBL81] R. LeBlanc and A. B. Maccabe, "PRONET: Language Features for Distributed Programming," Interim Technical Report GIT-ICS-81/03, May, 1981.

[LISK72] B. H. Liskov, "A Design Methodology for Reliable Software Systems," FJCC Proceedings, 1972.

[MACC82] A. B. Maccabe, "Language Features for Fully Distributed Processing Systems," PhD thesis (in preparation).

[MILL82] J. A. Miller and R. J. LeBlanc, "Distributed Compilation: A Case Study," Proceedings of the Third International Conference on Distributed Computing Systems, October, 1982 (to appear).

[PARN72] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems Into Modules," Communications of the ACM, Vol. 15, No. 12, 1972.

[PETE81] L. J. Peters, Software Design: Methods and Techniques, Yourdan Press, New York, 1982.

[SNOW78] R. A. Snowden and P. Henderson, "The TOPD System for Computer-Aided System Development," in Infotech State-of the Art Report on Structured Analysis and Design, Infotech, Maidenhead, England, 1978.

[SUTT81] S. A. Sutton and V. R. Basili, "The FLEX Software Design System: Designers Need Languages, Too," Computer, Vol. 14, No. 11, p. 95-102, November, 1981.

[WIRT71] N. Wirth, "Program Development by Stepwise Refinement," Communications of the ACM, Vol. 14 No. 4, April 1971.

[YAU81] S. S. Yau, C. C. Yang, and S. M. Shatz, "An Approach to Distributed Computing System Software Design," IEEE Transcations on Software Engineering, Vol. SE-7, No. 4, p. 427-435, July, 1981.

2.2 LANGUAGE SUPPORT FOR ROBUST DISTRIBUTED PROGRAMS

2.2.1 Why a Language for Distributed Applications?

Initially, high-level language work on distributed systems involved the development of "clever" compilers for traditional languages. These compilers partitioned the code produced for distributed execution in a way which was more or less transparent to the user. However, we feel that, since distributed systems involve new models of computation, it is appropriate to design new languages which provide primitives more suited to these new models. Examples of such new language features may be found in CSP ([Hoar78]), DP ([Brin78]), PLITS ([Feld79]), and ARGUS ([Lisk79] and [Lisk82]).

New features aiding the design and description of distributed programs are central to the design of PRONET ([Macc82]), a language currently being implemented at Georgia Tech. The new capabilities developed for this language are being added to Pascal as a base language, but since they involve only interprocess communication and interconnection of processes via message channels, they could be added to many other languages.

Among the important features of PRONET are the abstraction capabilities which it provides for the specification of networks. Network specification and process description are separated in PRONET by the division of the language facilities into two sublanguages: NETSLA (Network Specification Language) and ALSTEN (an extended Pascal for process description). These capabilities enable the interactions between processes to be encapsulated, aiding in the understanding of complex programs and providing information to the distributed operating system needed for making placement and scheduling decisions. A further description of these aspects of PRONET can be found in Appendix C.

PRONET also provides features which take advantage of the capability of distributed systems for graceful degradation. These features allow recovery from mechanical failures in a network. Appendix D provides an overview of these features.

2.2.2 Problems in the Design of PRONET

During the course of work on PRONET, areas for further study have been identified. In particular, our experience has shown that PRONET lacks facilities for providing the robustness in the face of algorithmic failures (due to flaws in software design), which is a desired property of distributed

programs for some applications. This problem, and proposed approaches for its solution, are examined in detail below.

Some other questions generated by the design and implementation of PRONET which have been identified are:

- The process description language ALSTEN, being based in Pascal, is necessarily simple. Better abstraction facilities, such as those provided by Ada, may be useful;
- Process execution is currently straightline sequential. The "actors" model of process execution may be more appropriate to the asynchronous nature of the interprocess communication model employed in PRONET;
- The NETSLA sublanguage needs more information about physical network attributes. Information about, say, the node at which a user is most often located, or which nodes are nearest to the user, would be useful in scheduling;
- The interprocess connections provided by NETSLA are currently unintelligent. The provision of pipe transforms and consistency checks in these interconnections should be attempted;
- Problems have been encountered in interfacing the PRONET implementation with the local operating system. In particular, the distinction between the command language of the OS and the NETSLA sublanguage (which is a command language of sorts) is hazy;
- The run-time support routines of PRONET at present subsume most of the functionality required of a distributed OS. The use of operating system primitives appropriate to the implementation of the new facilities provided by PRONET would be helpful. The implementation of the language on a distributed system running under such an operating system should be easier when attempted.

2.2.3 The Problem of Algorithmic Failure

Failures in distributed systems are of two varieties: mechanical (failures in system hardware) and algorithmic (failures due to software errors or design flaws). Schemes for dealing with failures have recently been surveyed by Kohler ([Kohl81]). As has been mentioned above, PRONET provides extensive features for aiding recovery from mechanical failures. However, the problem of algorithmic failure has yet to be addressed in PRONET.

Methods for treating algorithmic failure have been surveyed by Randell ([Rand79]). He divides these schemes into the so-called forward and backward automatic recovery schemes. In forward-recovery methods, predictions about the location and consequences of software errors are necessary, and thus these methods are not suitable for treating errors caused by design faults. The exception-handling methods used in languages such as PL/I and Ada are forward-

recovery methods, and are used mainly for anticipated errors or conditions such as faulty data or overflow.

Backward-recovery schemes, on the other hand, assume no previous knowledge of the location or nature of faults. Rather, backward recovery is analogous to mechanical backups in hardware systems. Information about the system state previous to the fault is restored from a checkpoint, and a back-up process is started. The back-up process is necessarily not the same as the failed process, as it would only fail again. In general, the back-up process (or processes) is more simple than the original process, and may provide only a primitive simulation of the functions of the original process (such as forwarding messages) in order to keep a network going.

The recovery-block scheme described by researchers at the University of Newcastle-upon-Tyne ([Shriv79], [Shriv81]) is an example of a backwards-recovery method. The syntax for describing a sequence of recovery blocks is:

```
assure <acceptance test> by
   <original block>
else by
   <back-up block 1>
else by
   ...
else error;
```

where some of the "back-up blocks" may be simple retries of previous blocks. If a failure occurs in the original block, back-up blocks are tried until one completes without failure and the acceptance test is satisfied, or else an error is signalled. The back-up blocks may have to undo permanent effects made by their predecessors before doing their own work.

Problems in the implementation of recovery blocks include the selection of checkpoint intervals and of appropriate points at which previously checkpointed information may be discarded ([Russ80]). The discarding of checkpoint information is equivalent to "commitment" to the results of the checkpointed block.

In another recent paper from Newcastle-upon-Tyne ([Cris82]), Cristian notes that a mixed strategy of on-units and recovery blocks is necessary to obtain highly reliable software.

A possible strategy which should be attempted in adding algorithmic-failure recovery mechanisms to PRONET is the notion of "overlaying" a back-up process on the address space of its failed predecessor ([refs??]). This scheme would have the additional advantage of allowing transparent replacement of existing permanent network processes. Old software could be replaced at an appropriate time (say, at a checkpoint) by overlaying a new version on the address space of the old software, without having to halt the entire program.

Considerable further study of the reliability issue is required. It is not clear at present whether extremely complex programming conventions will be necessary in the framework of PRONET to achieve reliability. Thus, it may be more appropriate to design completely new programming languages if such complexity is not considered acceptable.

2.2.4 Proposed Solution

Appendix F provides a more complete survey of software fault tolerance techniques and some proposed research directions.

2.2.5 Relationship to Other FDPS Work

Other major projects in the Georgia Tech FDPS project include the development of an operating system for an FDPS. The availability of a language which supports the clear expression of interprocess communication relationships and provides information for the partitioning of programs to execute in a distributed manner should prove quite useful in the implementation and operation of such an operating system. Also, it is clear that language facilities supporting transparent error recovery should be useful in the implementation of software, such as operating system components, which require high reliability, as well as to user programs.

2.2.6 Resources and Schedule

To cover an 18 month period:

Manpower	man-months
Senior Staff (3 m-m/year)	4.5
Junior Staff (6 m-m/year)	9
Programmers (6 m-m/year)	9
Secretarial Support (1 m-m/year)	1.5

Equipment

Computer Time Substantial

Timing

First period of 9 months:
Transport and complete current PRONET implementations; design new language features.

Last period of 9 months:
Implement and evaluate new features.

2.2.7 References

[Brin78] Brinch Hansen, Per, "Distributed Processes: a Concurrent Programming Concept," Comm. ACM 21, 11 (Nov. 1978), 934-941.

[Cris82] Cristian, Flaviu, "Exception Handling and Software Fault Tolerance," IEEE Trans. Comput. C-31, 6 (June 1982), 531-540.

[Feld79] Feldman, Jerome A., "High Level Programming for Distributed Computing," Comm. ACM 22, 6 (June 1979), 353-368. .RE [Hoar78] Hoare, C.A.R., "Communicating Sequential Processes," Comm. ACM 21, 8 (Aug. 1978), 666-677.

[Kohl81] Kohler, Walter H., "A Survey of Techniques for Sunchonization and Recovery in Decentralized Computer Systems," ACM Comput. Surveys 13, 2 (June 1981), 149-183.

[Lisk79] Liskov, Barbara, "Primitives for Distributed Computing," Proc. 7th Symp. Operating Systems Principles (Dec. 1979), 33-42.

[Lisk82] Liskov, Barbara, "On Linguistic Support for Distributed Programs," IEEE Trans. Software Eng. SE-8, 3 (May 1982), 203-210.

[Macc82] Maccabe, Arthur B., "Language Features for Fully Distributed Processing Systems," PhD thesis (in preparation).

[Rand79] Randell, B., "Software Fault Tolerance," Euro-IFIP 79 (P.A. Samet, ed.), 721-724.

[Russ80] Russell, David L., "State Restoration in Systems of Communicating Processes," IEEE Trans. Software Eng. SE-6, 2 (March 1980), 183-194.

[Shri78] Shrivastava, Santosh Kumar, and Jean-Pierre Banatre, "Reliable Resource Allocation Between Unreliable Processes," IEEE Trans. Software Eng. SE-4, 3 (May 1978), 230-241.

[Shri79] Shrivastava, Santosh Kumar, "Concurrent Pascal with Backward Error Recovery," Software - Practice and Experience 9 (Dec. 1979), 1001-1033.

[Shri81] Shrivastava, Santosh Kumar, "Structuring Distributed Systems for Recoverability and Crash Resistance," IEEE Trans. Software Eng. SE-7, 4 (July 1981), 436-447.

2.3 COMPILER DEVELOPMENT TOOLS

If distributed systems constructed of heterogeneous computers are to be useful as unified systems (rather than just as communication networks like ARPANET), the compilers available to programmers must be far more compatible with one another than those currently supplied by vendors. For example, compilers for the same language on different machines must accept exactly the same features and implement them consistently. Further, the user interfaces to these compilers should be consistent. These requirements imply that the development of a heterogeneous distributed system in which the differences among the machines are largely invisible to users will require the development of families of new compilers which are not machine-dependent any more than necessary.

We use the term "families" of compilers because of the techniques we envision for their construction. A family of compilers for a particular language will be said to exist on a distributed system when the compilers for that language on the various machines all use a common machine-independent front-end. (A front-end is that portion of the compiler which inputs the source program and translates it to some intermediate form (IMF).) This sharing of a front-end leads us to think of the group of compilers as a family.

The development of a compiler is usually considered to be a complex and expensive task. It will be necessary to build a powerful collection of compiler generation tools in order to make our idea of developing new compilers for a distributed system practical. The following sections describe some areas in which work might be done to facilitate the creation of such tools. The net result of this work should be a tool set which provides the maximum possible support for the generation of compilers usable on a distributed system.

Most of the work on this project will be software development rather than research. It should be possible to draw on many related but less comprehensive development efforts. Some of these are described in [Lanc76], [Cole74], [Gyll] and [Basi75]. The comprehensive compiler development system described recently in [Rudm82] even goes beyond many of the needs we anticipated, since it is oriented toward the development of extremely large programs.

2.3.1 Front-end Generation

We have already developed programs which generate table-driven parsers and scanners from formal syntactic specifications. Using these tools, it is quite straightforward to implement a machine-independent front-end of a compiler for any programming language. A single such front-end could be used as part of any number of compilers for a particular language running on different machines, thus taking an important first step toward solving the compatibility problems discussed above.

There are two tasks which would improve the utility of these tools:

Task 1: The parser and scanner generators should be combined into a single tool, along with some mechanism to support the automation of IMF generation as a result of parser actions.

Task 2: Practical front-ends require the inclusion of some mechanism to handle syntax errors discovered by the parser. A number of such mechanisms have been proposed and we have implemented one. An evaluation of the practical factors (such as time and space requirements) of the various alternatives should prove useful, in order to choose the appropriate technique for inclusion in our compiler generation system.

2.3.2 Automated Code Generator

In addition to the families of compilers previously described, another kind of family may also exist on a distributed system. If compilers for more than one language share the same IMF, then a single code generator can serve to finish the compilation job (translating IMF to machine code) on each kind of machine.

Considerable work has been done in recent years concerning the automation of code generation. While none of the results are commonly used in production compilers, considerable progress has been made. We would like to evaluate these efforts in order to see if any can practically minimize the work needed to generate the families of compilers we envision. This evaluation will include the implementation ofat least one such system.

2.3.3 A Multi-language Code Generator

Lacking any such automated code generation capability, we have constructed one code generation tool to facilitate compiler implementation on our PRIME computers. A general code generator which accepts a symbolic, tree-structured intermediate language is currently in use for the implementation of

several compilers. This program was designed to generate high-quality code by means of a complex case analysis. While no automated techniques went into the implementation of this code generator, its availability in a form which can be used by any compiler writer essentially eliminates dealing with machine-specific details from our compiler efforts. Similar generators for other machines would considerably simplify the envisioned compiler development efforts.

An attempt should be made to retarget this code generator so that it generates code for the VAX. From this effort, we should learn about machine-dependencies in the IMF it uses and we will obtain a capability to construct families of compilers for the VAX and the PR1ME which share a front-end.

2.3.4 Unification of Compiler Tools

The previous sections have discussed work involving a variety of compiler tools. All of the tools are based on separate theoretical developments and thus work in different ways. Compilation and all of its intermediate phases are basically translation tasks, that is, transforming some input language to a different output language. It thus seems feasible to build a higher-level compiler generation tool which would allow similar specification techniques to be used to automate the construction of as many of the phases of a compiler as possible.

2.3.5 Relationship to Other FDPS Work

No other FDPS development projects are dependent on this work. Rather, it is targeted toward enhancing the usability of an FDPS. The development of a unified set of tools would greatly simplify the construction of the required families of compilers.

2.3.6 Resources and Schedule

To cover a 12 month period:

Manpower	man-months
Senior Staff (3 m-m/year)	3
Junior Staff (3 m-m/year)	3
Programmers (2 at 6 m-m/year)	12
Secretarial Support (1 m-m/year)	1

Equipment

Computer Time Moderate

Timing

First period of 6 months:
Existing tools should be adapted to run on VAX and Prime.

Last period of 6 months:
Development of unified tool concept; implementation of automated code generators.

2.3.7 References

[Basi75] Victor R. Basili and Albert J. Turner, "A Transportable Extendable Compiler," Software-Practice and Experience, Vol. 5, pp. 269-278, 1975.

[Cole74] S. S. Coleman, P. C. Poole and W. M. Waite, "The Mobile Programming System, Janus," Software-Practice and Experience, Vol. 4, pp. 5-23, 1974.

[Gyll79] H. C. Gyllstrom, R. C. Knippel, L. C. Ragland and K. E. Spackman, "The Universal Compiling System," SIGPLAN Notices, Vol. 14, No. 12, December 1979.

[Lanc76] Ronald L. Lancaster and Victor B. Schneider, "Quick Compiler Construction Using Uniform Code Generators," Software-Practice and Experience, Vol. 6, pp. 83-91, 1976.

[Rudm82] Andres Rudmik and Barbara Moore, "An Efficient Separate Compilation Strategy for Very Large Programs," SIGPLAN Notices, 17, 6, June 1982. pp 301-307.

2.4 COMPILATION TECHNIQUES FOR DISTRIBUTED PROGRAMS

2.4.1 Introduction

In the previous section, techniques for implementing families of compilers were discussed. This section examines problems which result from the fact that these compilers must compile distributed programs. That is, programs will be constructed from separate parts, which will undoubtedly be compiled separately and may not ever all exist on a single machine. Thus compilation techniques must be developed for the language features which allow the specification of such programs. These new techniques may well involve interactions with linkers and with the operating system in ways quite different from what is common on current systems.

The implementation of PRONET will provide our first experience with these problems. The prototype implementation, currently in progress, will not deal with these problems in their full generality since we do not yet have a distributed operating system that can provide all of the necessary support. Once such an operating system is available, some effort should go into a study and implementation of separate compilation in a distributed system.

2.4.2 *Relationship to Other FDPS Work*

Any FDPS project which will involve the development of a distributed program will benefit from this work.

2.4.3 Resources and Schedule

To cover a 6 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	1
Junior Staff (2 m-m/year)	1
Programmers (6 m-m/year)	3
Secretarial Support (1 m-m/year)	1
Equipment	
Computer Time	Substantial

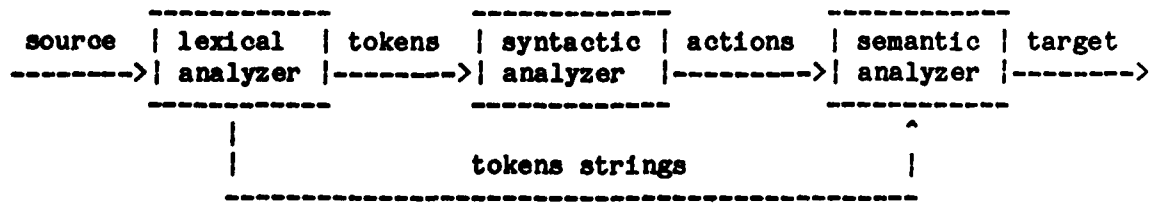
2.5 DISTIBUTED COMPILERS

To completely utilize a fully distributed processing system (FDPS), distributed versions of standard systems programs should be developed to take advantage of any inherent parallelism. Such distributed programs have the potential of greatly reduced response times. For the following reasons compilers are an ideal case in point: (1) Compilers have a high degree of inherent parallelism, (2) they are highly utilized system programs, and (3) they are currently slow in compiling large programs. We therefore propose to develop practical distributed compilers to be used in an FDPS.

2.5.1 Background

As a first step in this direction, a distributed compiler for a small language called Jigsaw was implemented (Jigsaw has if and while control structures, integer, real, array, and record data structures, and parameterized procedures). The purpose of this initial research was to test the feasibility of distributed compilation and to get a handle on potential response time improvements.

The implementation of this distributed compiler consisted of partitioning the compilation task into three subtasks, a lexical analyzer, a syntactic analyzer, and a semantic analyzer which includes a code generator. Code generation was included in the semantic analyzer because of the simplicity of the target code. In practical distributed compilers the semantic analysis and code generation would likely be split into separate subtasks. The three subtasks were implemented as distributed processes that communicated with each other by sending and receiving messages. The three processes along with their communication links constitute a pipeline where the source code is successively manipulated by each process to produce the target (object) code. Diagramtically the compiler looks like the following:



In structuring the compiler this way, the three processes can execute with a high degree of independence. For example, the lexical analyzer iteratively reads lines of source code, generates the corresponding token num-

bers and token strings, and sends them to the syntactic and semantic analyzers respectively. At the same time the syntactic analyzer will, as soon as it has token numbers in its receive queue, begin generating action numbers to send to the semantic analyzer. The semantic analyzer will be similarly executing at the same time. As long as the receive queues do not become empty or full the processes will not have to wait on each other. Under these ideal conditions, if we assume communications delays are negligible and that the three processes require an equal amount of processor time to perform their analyses, the response time on an unloaded system would be 1/3 that of a nondistributed compiler. Obviously, this is a substantial improvement, if it can be realized.

To test this distributed compiler, it was run on the School of Information and Computer Science's network of PR1ME computers. The lexical and syntactic analyzers were run on PR1ME 550's while the semantic analyzer was run on a PR1ME 400. The compiler was timed on test programs varying in size from 25 to 1200 lines of code and compared to the timing results of a nondistributed single pass version that was constructed from the same basic components. For an unloaded system we found the response time of the distributed compiler to be 1/2 to 1/2.5 that of the nondistributed version. For light to moderate loads the response time of the distributed version was about 1/1.5 that of the nondistributed version. These results clearly demonstrate the feasibility and desirability of distributed compilation. A more complete description of this work can be found in Appendix G.

2.5.2 Problems and Proposed Solutions

The next step is to develop practical distributed compilers for full scale languages such as Pascal. Such a distributed compiler would have the same basic structure as the one we implemented and could be constructed using the tools mentioned in section 2.3. However, three additional complications must be considered. First, the semantic analysis needs to be separated from the code generation using two processes instead of one. Second, the coordination of useful error messages from each of the component processes necessitates more interaction between the processes. Finally, in a real FDPS the execution of a distributed compiler would be under the control of a distributed operating system.

2.5.3 Relationship to Other FDPS Work

A practical distributed compiler will need to be interfaced with a distributed operating system. The distributed operating system should facilitate the initiation of a compilation with a simple command and should schedule the component processes on the appropriate processors so as to optimize some measure of system performance. If the distributed operating system could find enough available processors to quickly schedule all of the compiler's component processes, then response time improvements like those observed in our implementation of the Jigsaw distributed compiler in the unloaded case are quite possible in other situations. Thus, in its later stages, this research will depend on the work being done on global operating systems. In the meantime, such interaction is not necessary.

It is our hope that work with a distributed compiler will provide us with insight into the general problem of distributed software. This project should also include some effort toward generalizing the concepts used in constructing the distributed compiler.

2.5.4 Resource and Schedule

To cover a 12 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	2
Junior Staff (3 m-m/year)	3
Programmers (6 m-m/year)	6
Secretarial Support (2 m-m/year)	2

Equipment	
Computer Time	Substantial

2.6 SOFTWARE VERSION MANAGEMENT

A distributed software system consists of modules developed, updated, and maintained (perhaps simultaneously) by several people, possibly at different nodes in the system. A system-wide version control system is required in order to maintain consistency following modifications originating at any node and to permit the recovery of previous versions.

2.6.1 Basic Version Control System

As a minimum, the version control system should maintain current and previous versions of both source and object code. This provides the capability to recover from unfortunate modifications by returning to the unmodified version. A version control system is necessary if system maintenance is permitted to occur locally.

The distributed version control system will be implemented as a distributed data base. The extension of our basic software management system to provide these minimum capabilities for our FDPS should be accomplished without significant difficulty. The most important consideration involves the case when modifications to the system are taking place simultaneously at different nodes. If both changes involve the same module, there is little threat to system consistency. One or the other modification will be incorporated into the system. Difficulties arise, however, when the modifications affect different modules, especially if the modifications are proposed as different solutions to the same problem.

The minimum version control system can also be used to maintain other types of information throughout the system life cycle with additional benefits. Besides source and object code, the version control system can also maintain requirements specifications, design specifications, and documentation on-line. The maintenance of specifications on-line increases visibility, permits quick resolution of ambiguities, and makes easier the tasks of design, implementation, and maintenance from remote locations. Little additional extension is required to provide these capabilities, which are basically clerical.

With somewhat more effort the basic version control system can be expanded to function as an important tool both in development and maintenance. A fully expanded version control system is essential if modern programming and management practices are to be used at full potential. Use of the complete

system is described in the following section.

2.6.2 Version Control System and Development

The expanded version control system can be an essential part of the development process from requirements specification through testing. The first step in the development process is the preparation of a requirements baseline. Prior to agreeing upon the final software system requirements, the requirements specification should be shown to be

- compatible with the overall system design (which includes hardware and desired operations, as well as software)
- complete (so no modifications are expected)
- consistent
- testable (note that testing considerations begin early in the development)
- possible to be implemented on the selected hardware system (It should be possible to estimate resource requirements at this time. If the resources estimated exceed the resources available, either the requirements can be modified to specify a more modest system or the decision can be made to improve hardware capability.)

Once the requirements for the project under development have been fully agreed upon by review by all interested parties, modification of the requirements specification should only be performed through the change control process in the same manner that modification occurs during maintenance.

The presence of a baselined requirements specification as maintained by the version control system permits the establishment and maintenance of forward and reverse traceability from requirements through design to code and from requirements to validation testing. Requirements tracing permits early detection of errors during the design phase, ensures comprehensive testing, and improves maintenance.

The next phase in system development is the establishment of a preliminary design. This includes, as a minimum, top level software structure, data base definition, interface definition, scheduling criteria, and analysis of critical algorithms. The preliminary design, which is maintained in a standard format by the version control system, should identify the mapping from requirements, identify the groups or individuals responsible for each software element, define modular structure, and specify required processing resources, data base organization, and estimated resource budgets (schedule, manpower requirements, and computer resources). Before detailed

design and implementation begin, the preliminary design baseline should be demonstrated to be viable. The following should be met:

- All requirements have been allocated.
- Storage, execution timing, and accuracy estimates established for all modules. In later phases, monitoring of the accuracy of these estimates can be used to identify potential performance problems.
- Data base defined.
- Performance requirements can be met under operational loading conditions.

These practices are intended to prevent the development of a system which does not meet requirements. The early identification of performance problems permits system redesign to reduce performance or processing requirements to occur before coding is underway and modifications become more expensive.

In these phases the version control system is used to increase visibility by maintaining requirements specification, design documents, and performance estimates on line, easily accessible to developers at different locations. The version control system may perform checking to ensure that all design elements are traceable to the requirements specification and that all requirements are met. In addition, automated tools may be used to ensure that desired conditions, such as consistency and completeness, are met by the design.

Detailed design breaks down the preliminary design modules into routines and specifies implementation features. In this phase as well, the version control system may be used to ensure traceability.

When coding is underway, the version control system is used to maintain and update source code and documentation. Enforcement of programming standards is also performed to ensure uniformity and aid in maintenance. The version control system can include automated tools to verify that programming standards are met. Documentation standards should also be maintained.

Code maintenance is only one application of the version control system during implementation. Management visibility is significantly increased. The version control system permits easy determination of the status of each module (design complete, coded, tested, etc.). The status of the entire system can be obtained from this information and problem areas in the schedule

identified. Testing can also benefit. During requirements, design, and implementation, test data for each routine should be identified. This data is maintained by the version control system (and may be required by it). During checkout, this data is used to test each routine and the entire system. Test failures should be referred to the individual or group that originally proposed the particular test. The version control system should maintain testing status information and can be used to verify that all tests have been met.

2.6.3 Version Control System and Maintenance

The version control system maintains a program library which will frequently contain several versions of each routine. The controlled or system master portion of the library contains only those versions that form the operational system. Routines in this portion of the library should meet all project standards and be completely tested. Old, experimental, and test versions are also available in the program library, but must be clearly identified as such. During design and implementation, all software routines should be represented in the program library, at first as a program stub, then as untested code. During design, implementation, and unit test, modification of a routine should be permitted only by the individual or group responsible for its implementation. Following completion of unit test, routines in the uncontrolled section of the library can be modified by anyone as long as both old and new versions are maintained. Routines in the controlled portion of the library can be modified only with the approval of the change control board. The change control board must also approve deletion of uncontrolled routines. It is important that the version control system implement these restrictions by requiring appropriate authorization for any change.

The version control system is also used to expedite the processing of change requests. When a change request is received, the version control system notes the source of the request and the date and records the request. The change control board, which may be a single individual in charge of maintenance if the system is small, or may consist of representatives from system design, operation, management, and users in the case of a large system, obtains change requests from the version control system. For each request, the board must consider its necessity, its priority relative to other needs, its possible side effects, and availability of funding. The board may decide that a requested change should be implemented at once, scheduled for

implementation, deferred for further analysis, deferred without analysis, or rejected. The action taken should be reported to the originator of the change request by the version control system which also maintains the status of all requests until the requested change has been completed to the controlled system (at which time the originator of the request should be notified) or rejected by the board. The board uses the version control system to determine possible side effects of changes by tracing back to the requirements specification and forward to code. The version control system may notify the change control board that a requested change has been repeatedly deferred or that the same change has been requested by more than one originator.

When a change is authorized, the board assigns responsibility for development and testing to groups or individuals, sets the schedule, and specifies the budget. The version control system permits the board to modify status of authorized changes and to take additional action if required. Status information may also be available to the originator of the change request.

The version control system should permit any changes to be "undone" by backtracking to a previous version. Maintenance of previous versions permits this capability, but backtracking must be done carefully with consideration of possible interactions between changes. Two requested changes can result in four versions: the original, modification 1, modification 2, and both modifications (1 and 2). If the changes occur successively, three versions will be present in the system, the original, modification 1, and both modifications. If it becomes necessary to undo modification 1 by backtracking to the original routine, modification 2 is also undone. This should be reported by the version control system, which must therefore maintain an audit trail of all authorized changes.

2.6.4 Relationship to Other FDPS Work

The considerations outlined in the preceeding sections apply to the version control system for any large software project, distributed or not. A few additional considerations can be identified for the distributed version control system.

A distributed software system must be able to operate in varied environments. If the processors comprising the distributed system are homogenous, the environments may vary in loading and in available resources.

The software system must be able to meet performance and resource usage requirements in each environment. The presence of processors that are not identical increases environment variability. Use of a high level language may permit one version of the system to operate at all nodes; however, particularly for routines with high performance requirements or high machine dependency, it may be desirable to maintain separate versions for the different environments. If this is done, extra care must be taken to ensure that all versions meet requirements and to ensure that all modifications are carried out correctly.

Change requests may be local or global in nature. For instance, rounding conventions on one system may cause errors that do not occur at other nodes. A change to correct this problem will typically affect the system only at the node at which it occurs. It may be considered desirable to permit such changes to be authorized and carried out locally, or they may be authorized by the global change board and implemented locally. In either case, local modifications should be subject to the same traceability requirements and standards as global modifications and should be available to the global version control system. If this is not done, system behavior can become unpredictable whenever locally modified routines are later modified globally.

2.6.5 Resources and Schedule

The extension of our basic software management system to provide the capabilities of the basic version control system for our FDPS should be accomplished without significant difficulty. Development of the expanded distributed version control system will require significant effort and may be undertaken as a series of modifications to the basic system. We believe that the benefits resulting from the increased capabilities more than outweigh the additional effort.

To cover a 12 month period:

Manpower	man-months
Senior Staff (3 m-m/year)	3
Junior Staff (3 m-m/year)	3
Programmers (12 m-m/year)	12
Secretarial Support (2 m-m/year)	2

Equipment

Computer Time	Moderate

Timing

First period of 6 months:
Extension of basic version control system

Last period of 6 months:
Development of expanded version control system

2.7 COST ESTIMATION FOR DISTRIBUTED SYSTEMS

Cost estimation for distributed system development closely parallels standard models of cost estimation. It is expected that the weightings of various factors may require adjustment to represent distributed development more closely. In particular, system integration can be expected to require a significantly larger portion of development resources, particularly if development is carried out simultaneously at different nodes. Manpower estimates must be accurate at the modular level to permit development of modules in different locations.

Correct, early estimation of resource requirements and the assignment of available resources to various system components is particularly important in the development of distributed software, as system loading may be affected. Correct, early estimation of resource requirements and the assignment of available resources to various system components is important to any large software development. If resource requirements can be estimated early, it is possible to identify possible performance failures before coding takes place. This is discussed further in the section on version control.

The first step in developing a cost estimation system for our FDPS is identifying and obtaining a standard cost estimation system. The system chosen should permit detailed estimation of resource and manpower requirements at an early phase. The systems developed by Putnam and by Boehm are possible candidates. Tuning of the standard system for correct estimation in a distributed environment will be accomplished by maintaining careful resource usage and manpower scheduling records for all software developments. The model tuning process is not expected to require significant effort, but will not be possible until a number of software development projects have been completed.

Lacking expertise in the cost estimation area and significant experience in FDPS software development, we have only been able to identify this problem as one requiring further study.

SECTION 3

DISTRIBUTED SYSTEM DESIGN SUPPORT FACILITIES

3.1 INTRODUCTION

As defined in Section 1 of this report, the primary purpose or function of the System Design Support Facilities is providing information such as performance, inherent reliability, etc. about the system, its design, or its implementation. This is accomplished by the use of simulators, emulators, monitors, estimators, and testbeds as well as combinations of all of these. This class of support capabilities is implemented in software alone as well as with combinations of special hardware and software.

Everyone has a list of favorite support capabilities in this class. These lists seem to be heavily influenced by that individual's experience in using one or the other. There has not been a great deal of study of this area, so the list below is certainly influenced by our own experiences in studying and implementing distributed systems. We have attempted to include several that we have not had firsthand experience with, but that coverage is probably quite incomplete.

- Accurately modelling/describing the system under examination.
- Validity and accuracy of the information obtained by either direct or indirect measurement.
- Ability to obtain information without "distorting" the operation of the system.
- Obtaining the information in a timely and efficient manner.
- Cost of the support facility.

The major problems in the development, implementation, and use of any members of this class of support facilities are common for almost all of the different capabilities.

3.2 PERFORMANCE MEASUREMENT

3.2.1 Purposes of Performance Measurement

There are three general purposes of performance evaluation: selection evaluation, performance projection, and performance monitoring [Lucas 71]. These are shown in Figure 3-1.

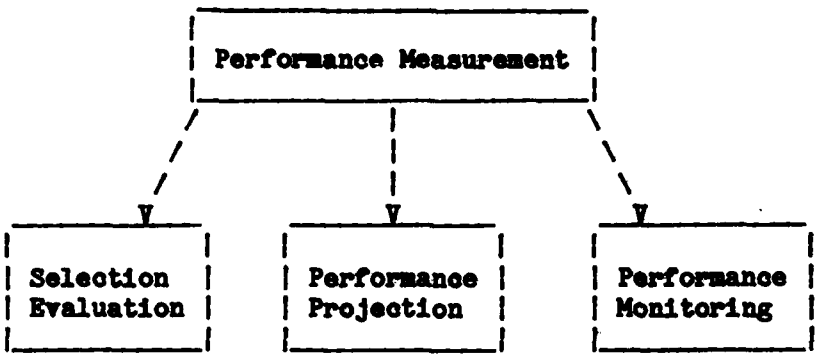


Figure 3-1 —— Purposes of Performance Measurement

Selection evaluation involves the comparison of existing systems. The most frequent application of selection evaluation techniques is for comparison of computer systems to determine which system performs a given function most efficiently or whether a given system configuration can support a particular application. Selection evaluation is also applicable when measuring the impact of different hardware or software on an existing system. For example, selection evaluation is useful in determining whether the addition of a load balancing algorithm improves interactive response time. Similarly, selection evaluation can answer the question "Did the last change to the operating system improve performance?" In all cases, the defining feature of a selection evaluation is that the systems to be compared must exist and must be available for testing.

Performance projection techniques are often applicable during the design of new hardware and software systems. These techniques attempt to predict the performance of new hardware and software designs prior to implementation. They can also be used to predict the performance of a system under a new workload or with a different hardware configuration. Performance projection techniques can often be applied to the same problems as selection evaluation techniques. However, the distinguishing feature is that it may not be practical

to actually test the systems under consideration: it may be too expensive to test the actual configuration, the configuration may not be available, or the system may not exist at all.

Performance monitoring techniques are applied in an attempt to understand the behavior of existing systems towards the goals of improving efficiency and service to users. It usually involves observing an existing system under normal operating conditions. Quantities measured with performance monitoring techniques are usually very dependent on the system measured (e.g., number of page faults, number of times the dispatcher is entered, etc.). For this reason, performance monitoring techniques are usually applicable only for the comparison of similarly structured existing systems. For instance, it is difficult to compare the performance of systems that use different disk block sizes by comparing the number of physical disk reads and writes.

In a distributed processing system testbed facility, performance evaluation will be necessary for all three purposes. One need for a performance measurement tool is in the area of selection evaluation. It is necessary to test prototype systems and compare the results with the results predicted by performance projection techniques, as well as with results obtained by testing other systems. The tool must be able to empirically measure the performance of existing software and hardware configurations, and must be able to provide comparable measurements on similar configurations.

3.2.2 Techniques for Performance Measurement

A number of different performance measurement techniques can be applied for the purposes mentioned above. Figure 3-2 shows these techniques.

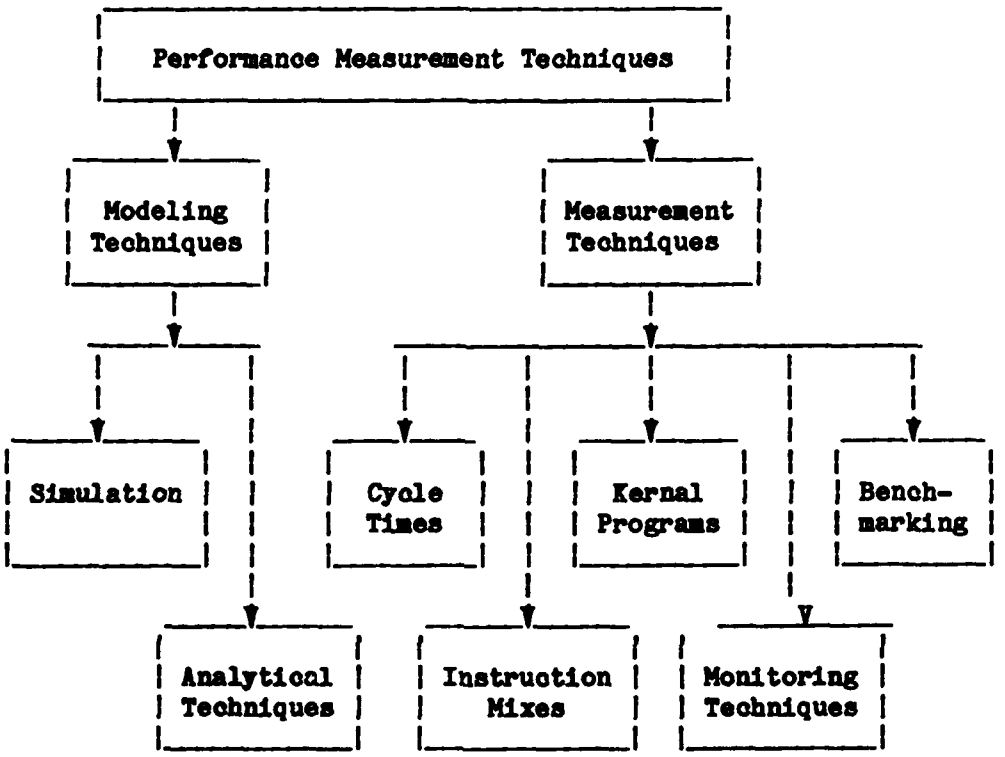


Figure 3-2 —— Techniques for Performance Measurement

Most of these techniques can be utilized for all purposes of performance measurement, but some provide only marginally useful results. Since a distributed processing system testbed performance measurement tool is needed for the purpose of selection evaluation, the following discussion of performance measurement techniques is confined to those applicable to selection evaluation.

There are two classes of performance evaluation techniques that can be used for selection evalution: modelling techniques and measurement techniques [Ferrari 78]. Modeling techniques involve building a representation of the system to be evaluated and then testing that model. Although most useful in performance projection, modelling techniques can also be used for selection evaluation. A significant problem with all modelling techniques is determining how well the model reflects the system it models.

> Validation[of a model] is often difficult, and sometimes impossible. It may be based on previous theoretical or simulation results, but if the modeled system exists, the ultimate foundations of a validaton model must be empirical. . . Thus, in a sense, measurement is the most important evaluation technique, since it is needed also by the other techniques. [Ferrari 78]

Measurement techniques involve actually measuring the behavior of an existing system and are thus applicable only when the performance of a system can actually be determined. Several of the measurement techniques (instruction timings, instruction mixes, and kernel programs) merely make comparisons of hardware parameters such as memory cycle time, addition times, etc. These techniques are generally useful only as a supplement to more powerful techniques when used to compare hardware configurations and are inadequate when used to compare software systems [Lucas 71].

Hardware and software monitoring techniques, which usually involve the recording of such things as the number of page faults, number of cache misses, etc., provide a great deal of information about the performance of a particular system. But since the parameters that can be measured are usually very specific to a particular implementation, comparisons between systems with different internal structures are usually difficult to interpret.

The remaining measurement techniques, generally called benchmark techniques [Svobodova 76], involve actually running a system using a set of real or carefully contrived input and measuring the response of the system. Since the benchmark techniques treat the system under test as a "black box", measuring only stimuli and responses, they are immune to many of the problems of other measurement techniques. In general, the only significant difficulties of benchmark techniques are in the determination of the input to the system under test and in the analysis of the output of the system under test.

To support the testbed, the performance measurement must be capable of consistently applying arbitrary benchmarks to the machines that are or will be a part of the testbed. It must also allow arbitrary analysis of the responses of the testbed equipment. This decision permits a generally useful tool for the testbed, while not encumbering or presupposing knowledge of the research issues of either the FDPS project or of benchmark techniques.

3.2.3 References

[Ferrari78] Ferrari, Domenico, Computer Systems Performance Evaluation, Prentice-Hall, 1978.

[Lucas71] Lucas, Henry C., "Performance Evaluation and Monitoring", Computing Surveys, Vol. 3, No. 3, 1971, pp.79-91.

[Svobodova76] Svobodova, Liba, Computer Performance Measurement and Evaluation Methods: Analysis and Applications, Elsevier North-Holland, Inc., 1976.

3.3 SIMULATORS

3.3.1 Description

Simulators are perhaps the most popular kind of facility used to support the design of distributed processing systems. They represent the research technique of first choice (as well as often the last resort) for answering many questions about the operating characteristics of proposed systems. These facilities range in scope and complexity from fairly short computer programs designed to answer specific questions, to quite substantial software systems capable of addressing a broad range of problems and a variety of operating environments. In any digital simulation, there is an attempt to model the salient features of a target system with repesentations of its current status and the significant events affecting that status. Simulators are generally designed to model the interrelationships among many subcomponents of a system in such a way that their interaction and the effects of various operating parameters on the overall performance of that system can be examined and recorded.

Network simulators are tools for modelling network component interactions. They are essential in the early analysis and design phases of network development and extremely useful during the maintenance of such a system. The purpose of these simulators is performance measurement and evaluation of communication protocols, network control algorithms, network topologies, and many other operational characteristics. The simulator program executes a sequence of defined states or events in network activity. Input to the program are various parameters of the model. Output depends on the function of the tool. The output could be a transaction log of an event sequence, for example, or performance measures of critical events.

Typically, the simulation tool models a manageable subset of the overall distributed environment. A hybrid simulation approach, combining network simulators with analytic models, is useful in modelling complex environments. Techniques like the hybrid approach may be the only way to represent in a simple form the complex interactions of an operational system. The hybrid technique is often useful in reducing the execution time requirements; however, it is only applicable when validated analytic expressions are available to describe the performance of well-defined components of the overall system.

3.3.2 Background

Interest in simulation techniques is both longstanding and widespread. Their application to computer systems seems particularly appropriate, in that the behavior of extremely complex target systems can generally be broken down into more manageable components which exhibit well-defined state-event transitions.

The development of simulators is largely motivated by obvious limitations in two prominent alternatives: mathematical analysis and empirical observation. Analytic techniques are often difficult to understand, requiring a fairly extensive background in applied mathematics for adequate comprehension. This limits not only the number of active practitioners, but also the population of appreciative and accepting readers. Far more serious perhaps are the simplifying assumptions required by many analytic models. It may be difficult to determine the extent to which these assumptions can be violated without completely invalidating the results of an analysis. When restrictive assumptions are weakened, an analytic model may become intractable, yielding only approximations, probabilistic algorithms, or very costly "brute force" computations. While the validity of any analytic or modelling technique to real systems is always suspect, this problem seems particularly serious for mathematical analysis.

Research data from empirical observations can be very persuasive in any scientific study. In asserting the merits of some new technique, it is particularly important to be able to compare measured performance with similar data based on observations of some other approach. Unfortunately, operational systems can be very costly to develop strictly for research purposes. The time required to construct such a testbed may be an even more serious consideration.

While simulators are certainly not a substitute for the development of prototypes, they can play an important role in the design process, before more substantial resources are committed. Additionally, they can provide an important research advantage over operational systems in offering greater control over extraneous variables that affect performance. In an operational system, it is usually impossible to eliminate or even hold constant all of these factors, particularly if the costs of research are controlled by using the testbed for other purposes. Perhaps even more serious are limitations on

the range of factors that are the primary target of investigation. To be able to generalize ones findings beyond the context of current operating parameters, it is important to be able to select extreme values for, say, demand for service or applied load. Various combinations of values may never occur naturally at a particular installation, or they may occur so rarely that interesting cases cannot be properly studied.

Modeling is an efficient manner of assessing system behavior. In the FDPS environment, network simulators extend modelling to cover a broad range of activities. Two major functions of network simulators are:

- Validate analytic models of operation.
- Evaluate performance of network protocols:
  - Components
  - Protocols
  - Interfaces

The assumptions of analytic models are constraining but necessary for adequate solutions. Message independence is an example of one strong assumption used in most analytic solutions. Also, most analytic results apply to a steady state condition, even though it may never exist. [Reiser82] Simulation is more flexible in describing such events, and the results may confirm the simplifying assumptions of analytic models.

The probabilistic character of network events and the complexity of their interactions places heavy demands on analytic models. Advances in queuing theory have not produced computationally tractible solutions to many problems. Simulators can evaluate network protocols that are analytically intractible.

Network simulators are an integral part of communication system support. The design, test, and maintenance stages of network development rely on simulation to evaluate various protocols, models of behavior, and computer communication products. Some of the areas of performance evaluation include:

Low-level communication protocols
Communication access methods
High-speed local area networks
Routing and flow control models
Distributed control models
Workload management algorithms
Transaction processing

As the number of computer networks increases, overall system performance evaluation under different workloads will guide maintenance and expansion design decisions.

3.3.3 Problems to be Solved

- Major problems to be solved
  - Ability to examine detail at various levels
  - Efficiency of run time
  - Validation
  - Ease of contruction
  - Modifiable

It is unfortunate that most simulators are designed as ad hoc facilities that cannot be readily integrated with or even compared to each other. This problem can probably be attributed to a very short software life-span, since simulation programs are often shelved after serving their limited research purpose. They rarely enter a phase of prolonged utilization that might justify an additional investment in standardization or even the kind of documentation that is expected for commercially viable software systems.

Validation of a simulation model is an even more serious problem, since the lack of it can adversely affect the credibility of any study. The difficulty lies in the selection of standards or criteria on which to base a validity assessment. Ideally, one might simulate a few limited cases which can be verified by agreement with available systems, but simulators are often developed precisely because the opportunity to test interesting cases does not exist on available systems. Agreement in the limited cases that do apply may be better than nothing, but not much, since simulators rarely exhibit operational uniformity. In fact, they often execute entirely different procedures to model very different systems. Similar arguments apply to validation with respect to analytic models. The further the simulator departs from the domain of these models, the less useful they are as a basis for validation.

The only satisfactory solution to this problem seems to be validation by independent simulation. General agreement of two or more simulators designed separately to study the same problems is an impressive achievement that reinforces conviction in the results of both. Ironically, it may be desirable in this context to intentionally limit the transfer of internal documentation for

each simulator, so that subtle artifacts do not cross over and contaminate the "independent" results.

Unfortunately, independent verification of previous experimental evidence is seldom seriously undertaken. The excitement and prestige of breaking fresh ground seem to draw attention away from the important work of confirming and consolidating previous findings. This, of course, is a general problem common to many scientific disciplines.

Simulators tend to be enormous programs. General problems of large scale software development relate to building an efficient simulator. Because of the large task of designing, documenting, coding, debugging, and testing a simulator, it may be economically more feasible to actually implement some network components in an existing testbed environment.

Often the statistical results of certain variables in the simulator are important to research. Classical statistics requires large numbers of independent samples to arrive at acceptable confidence levels.[Tobagi78] Generating a large sample size could involve thousands of computer runs of the simulator.

If the program is long, it may drive the computing cost of a large sample size extremely high. On the other hand, an experimental design based on a small sample size does not achieve critical levels of statistical significance and is generally unacceptable.

Other specific problems of network simulators are equally important. First, establishing the functions of the simulation tool and modular program specifications. Since the simulator cannot reproduce all the component interactions of a real system, some aspects of network behavior must be left out of the model for simplicity.

Second, developing simulation models of networks with a high degree of autonomy between nodes. The fully distributed environment relies on site autonomy to achieve many of the design goals. The autonomy complicates the component interactions to a degree that current efforts have failed to model them adequately.

Third, insuring that the program actually models the system in question and that the results of exercising the model are valid. When a real network exists, instrumented performance measures give an objective verification of

simulated results. On the other hand, experimental research often has no operational system to depend on for that kind of support. Under those conditions, the validity of the model depends on the internal consistency of the program, the design logic, and the results of other simulators. [Kobayashi78] Nevertheless, the validity of simulation models is an unresolved issue.

3.3.4 Proposed Solutions

To facilitate the modular development of a simulator, it seems appropriate to begin with the identification and organization of those alternatives relative to the concept of "service", perhaps along the lines suggested by the ISO Basic Reference Model for Open Systems Interconnection. A simulator could then be developed to evaluate the effects of these design alternatives and their interaction on performance goals for Fully Distributed Processing Systems.

Initial approaches to network simulation in the FDPS environment may concentrate on the modular design of the program or a collection of interacting programs to model system behavior. Dividing the overall system into a framework of independent but interrelated parts allows the design process to focus on the specific functions of each subsystem. Designing each subsystem individually and defining the interaction between modules will go a long way toward creating an overall model that is accurate and flexible.

One advantage of implementing the network simulator as an aggregate of subsystems is that this approach allows the use of a hybrid approach. Components of the system having analytic solutions can utilize those models, taking advantage of their simplicity. The mathematical results of these modules are integrated into the descriptive simulation procedures of other subsystems that are analytically intractible or require more detail.

Another advantage is that each subsystem module may run separately from others for more frequent trials. Achieving a significant sample size may be possible by running many smaller programs rather than one large program.

3.3.5 Relationship to Other FDPS Work and SSC's

Related work along these lines in the Georgia Tech FDPS Research Program has prompted the development of several different simulators. Each provides only a fairly limited view of a complete system, utilizing quite different assumptions about the underlying interconnection and support structure. This work needs to be expanded and consolidated into a more comprehensive evalua-

tion of options available at many different levels of analysis.

An event-based FDPS simulator has been developed which simulates functions typically provided by local operating systems, functions provided by a distributed and decentralized control scheme, and the load placed upon the system by users attached to the system through terminals. This simulator has been used successfully in two separate research efforts in the FDPS project: one analyzing control strategies, and the other analyzing work distribution schemes.

Most of the FDPS research relies on network simulators to compare and contrast different solutions to the unique problems we are confronting. Recent work at Georgia Tech in distributed processing involves simulation studies. [Martin80], [Enslow81], & [Sharp82] Simulation is the only method of evaluating the behavior of distrbiuted network activity in absence of an operational testbed. Further research on the operational support capabilities will need good simulation tools for adequate design and analysis.

3.3.6 Resources and Schedule

Such a project would probably require at least a one-third time commitment by three or four system analysts/programmers under the active guidance of two system designers for a period of approximately two years. This time period would permit evolutionary development, which is recognized as the only viable approach. Work should begin immediately, since the results of this study should be quite valuable in guiding other efforts in the development of an operational distributed processing system.

Considering the development of network simulators as a major software engineering task, many of the same resources are needed as for implementing an operating system. Basically the job requires computing services, time, and people. Computer resources for running simulations and analyzing the results are also necessary.

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 at 3 m-m/year)	12
Junior Staff (3 at 4 m-m/year)	24
Programmers (2 at 6 m-m/year)	24
Secretarial Support (3 m-m/year)	6

Equipment

Computer Time	Very high

Timing

3.3.7 References

[Enslow81] P. Enslow, T. Saponas; "Performance of Distributed and Decentralized Control Models for Fully Distributed Processing Systems," Georgia Institute of Technology, July 1981.

[Kobayashi78] H. Kobayashi; Modeling and Analysis, An Introduction to System Performance Evaluation Methodology, Reading, Ma.: Addison-Wesley Publishing Co., 1978.

[Martin80] E. Martin; "Operational Survivability in Gracefully Degrading Distributed Processing Systems," Georgia Institute of Technology, Dec. 1980.

[Reiser82] M. Reiser; "Performance Evaluation of Data Communication Systems," Proceedings of the IEEE, vol. 70, no. 2; Feb. 1982.

[Sharp82] D. Sharp; "Work Distribution in a Fully Distributed Processing Systems," Georgia Institute of Technology, Jan. 1982.

[Tobagi78] Tobagi et al; "Modeling and Measurement Techniques in Packet Communication Networks," Proceedings of the IEEE, vol. 66, no. 11; Nov. 1978.

3.4 LOAD EMULATORS

3.4.1 Remote Load Emulators - Short Description

One of the goals of the on-going research in the School of Information and Computer Science is the creation of a testbed facility for the implementation and evaluation of fully distributed processing systems (FDPS). An essential feature of the testbed is the requirement to empirically evaluate the performance of fully distributed processing systems during their implementation. Providing a facility that measures these systems by generating an external load and measuring external response can be done by a remote load emulator.

A remote load emulator (RLE) is a device that emulates sources of on-line input to a computer system. An RLE is one of the most reliable tools for measuring the performance of remote-access computer systems. The general purpose RLE must emulate both batch input and interactive sources. When the definition of interactive users is extended to include processes interacting with one another, we see that "interactive users" are of primary concern to us. In order to emulate a wide variety of interactive input devices, an RLE is controlled by programs known as scripts. A script describes a sequence of actions to be performed by the RLE. Such a sequence might include messages to be transmitted to the system under test along with their timing, responses possible from the system under test, and actions to be taken after a specific response is received. As well as performing actions as specified by the scripts, the RLE should record all the communication activity for later analysis.

3.4.2 Remote Load Emulators - Background

Performing a benchmark on a system first involves devising a workload to apply to the system under test. Svobodova defines the workload of a system as "the total of resource demands generated by the user community" [Svobodova76]. Seen from the benchmark point of view, devising a workload is simply defining the set of inputs to be presented to the system under test. It is not a function of a remote terminal emulator to design the workload to be used as the benchmark. The user must be responsible for devising a representative workload based on the system to be tested - the performance measurement tool need only be able to apply an arbitrary but defined workload.

Once a set of benchmark jobs have been chosen and tested, the workload can be applied to a particular system configuration. A batch system may be tested by simply placing each job deck in the card reader at a preappointed time, and noting the time needed for the completion of all of the jobs. Testing a slightly different configuration presents no additional problems. The workload in this case is repeatable; it can be run several times on one system and barring malfunctions, one can expect similar results.

Testing of an interactive system is much more difficult. Since an interactive workload is generated by users entering data at terminals, it is very difficult to generate a repeatable workload without additional computer assistance. In general, it is not possible to get a dozen or more people to type in commands in exactly the same order and "think" for exactly the same time for many consecutive test sessions. To obtain comparable results from several test sessions, it is necessary to have a means to emulate the actions of the interactive users and to repeat the same workload many times without tiring.

A Remote Load Emulator (hereafter referred to as an RLE) is just such a device. Its primary function is to emulate the load placed on a system by remote sources attached through communications links, such as terminals, sensors, and process controllers. RLEs are quite useful in performance measurement and evaluation, as well as for emulating devices in multi-dropped line protocols, monitoring communication line activity, and providing a host system for the testing of communications line protocols.

When used for performance evaluation, the RLE must produce a predefined workload while recording data about the responses of the system under test. To be capable of generating an interactive workload as well as a batch workload, an RLE must be able to accurately emulate people typing at interactive terminals. An interactive session, as opposed to a batch job, has three additional characteristics: 1) future input may be determined by current output, 2) there may be pauses before input messages corresponding to user "think time", and 3) there are pauses between input characters corresponding to user typing rate [Svobodova76].

For the needs of the distributed processing testbed, a remote load emulator is the best choice for the performance measurement tool. As a minimum, the RLE must be able to generate interactive workloads to drive the

existing hardware and software in the testbed. Preferably the RLE should be a general tool for performing benchmarks; it should be able to emulate any interactive device, either computer system or teminal, that hardware considerations allow it to replace.

3.4.3 Remote Load Emulators - Problems to be Solved

From the preceding discussion of the motivations for the RLE, two design objectives arise: the RLE should produce realistic interactive workloads and the RLE should remain an effective tool for several years. These objectives, although succinct, are not absolute requirements. It is necessary, as in most software projects, to compromise some of the objectives for practical reasons. For instance, extremely accurate time interval measurement cannot be provided without hardware modification. Requiring special hardware reduces the long-term usefulness of the RLE, but increasing its timing accuracy allows the generated workload to be more representative.

Two requirements are necessary to ensure the RLE's ability to generate realistic workloads: the RLE must be able to accurately emulate remote devices, and the workload presented by the RLE must be repeatable [Watkins 77]. These requirements are based on the primary motivation for the project: some method must be provided to accurately simulate real interactive users.

To be able to accurately emulate remote devices, the RLE must be capable of three things: it must be able to alter its behavior based on data it receives from the system under test, it must be able to accurately control delays between characters, and it must be able to accurately control delays between a response from the system under test and the next message from the RLE. These requirements follow directly from the defining characteristics of interactive workloads mentioned above.

The necessity that the RLE produce a repeatable workload is a direct result of the purposes for which the RLE will be used. Since it will be used to compare different hardware and software configurations, it must be capable of generating the same workload time and again. This is not to say, however, that given the task of generating the same workload, the RLE will generate identical output. If the behavior of the system under test differs, of necessity, response of the RLE will differ. What must be expected is that "each time the RLE presents an activity to the SUT [system under test] the observed performance differences are due to the SUT and not to the RLE" [Watkins77].

The requirements to ensure the long-term effectiveness of the RLE are perhaps more obvious, since they apply to most software systems as well. These include ease of use, ease of maintenance, and flexibility. It is clear that implementation of the RLE will be wasted if use of the RLE requires as much effort and knowledge as is required to implement a special program to be used once to perform the same actions.

The RLE will not be useful if it is not easy to maintain (e.g., if it requires a non-standard environment with its own special operating system and dozens of control files). Again, it will be pointless to keep the RLE if it requires more effort to maintain than it does to implement the special purpose programs the RLE replaces.

Finally, although the RLE must be easy to use, it must be flexible enough to perform complex and varied emulation tasks. A priori restrictions must be avoided that prevent the RLE from performing such tasks as simulating interactive devices other than user terminals, generating workloads for machines other than those in the testbed, posing as one or several terminals on a multi-dropped communications line, passively monitoring activity on a communications line, or emulating a host system for testing communications line protocols. The RLE must also be efficient enough to provide a number of concurrent sessions. Othewise, the RLE will be of little use in monitoring even the existing systems.

3.4.4 Remote Load Emulators --- Proposed Solutions

It is clear that the RLE must be able to support multiple concurrent interactive sessions, so some concurrency will be required in the RLE. The multi-user operating system supports multiple concurrent processes and virtual memory, while the single-user operating system does not. There are only two possible advantages in using the single-user operating system, assuming multiple processes are simulated to provide the necessary concurrency: code can be shared between processes, and process switching time can be minimized. These advantages are not significant though, since most modern multi-user operating systems allow reentrant code to be shared between processes.

Since use of the single-user operating system provides no obvious benefits and because it would noticeably complicate the project by requiring the implementation of process scheduling and concurrency primitives, use of the multi-user operating system is probably the best choice.

Another area for choice is the structure of the RLE itself. There are three different structures that can be used for the RLE: the RLE can directly interpret a human-readable script during the emulation session, the RLE can compile a human-readable script into a machine language program, or the RLE can compile the human-readable script into an easy-to-interpret intermediate form for execution. The principle difficulty with the first choice is that it takes a great deal of time to parse a free-form program. Since the number of simultaneous interactive sessions that can be run may well be determined by CPU time requirements, it seems foolish to place the parsing load in the most time-critical area when better alternatives are available.

The second approach, compiling a script into machine language, solves the objection to the first approach by allowing a complex script language while allowing quick execution. It does, however, present two other problems. First, it does not allow the sharing of code between scripts (except between identical scripts), since each script would be a separate object program. Second, it would significantly complicate the implementation to directly generate machine code, and generating assembly language or Fortran would inconvenience users by requiring a great deal of time for compiling and linking the script programs.

The last approach, compiling scripts into an intermediate form, minimizes the deficiences in both of the previous two approaches. It permits a complex source language, while permitting efficient interpretation. It also allows the interprete code to be shared among the concurrent processes and is much easier to implement and maintain. It is this approach that was used.

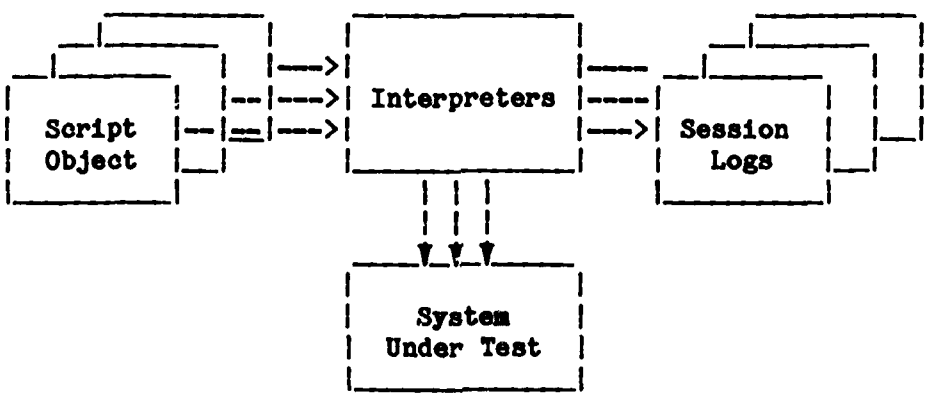
A difficult area to address is the analysis to be done on the output from an RLE test session. Little is known about what information will be required in the analysis of a test session, since many of the projects that might use the RLE have not been devised. Because of this, it is necessary to defer the decisions on the exact kinds of analysis that can be performed. Fortunately, there is an approach which allows this quite simply. The RLE time-stamps and records all input and output from interactive sessions during emulation. Instructions are written in the script to place various markers in this log along with the session transcription. Then, after the emulation session is complete, these logs can be analyzed. Since events of interest to the investigator have been tagged by markers in the log, time intervals can be

easily computed, and other information can be derived as needed. This approach has the benefits that the analysis code is not built into the RLE and can thus be changedwithout danger to the integrity of the RLE code, and since a complete record of the emulation session is made, analyses may be run and rerun on the same session without the need of repeating the expensive emulation session.

As discussed above, RLE contains three components: the preprocessor, the interpreter, and the analyzer. A diagram of the structure of the RLE appears in Figure 3-3.

Preprocessor:

Interpreter:



Analyzer:

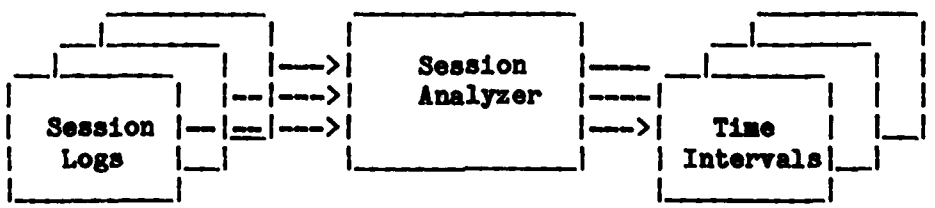


Figure 3-3 --- Structure of the RLE Implementation

3.4.5 Relationship to Other FDPS Work and SSC's

The development of a suitable RLE(s) will greatly enhance the value and usability of a testbed. It is not essential; however its value clearly outweight its costs.

To generalize the use of the RLE it must be able to emulate embedded processes interacting with one another. The ability to add this capability should be considered when designing the script defining language and preprocessor.

3.4.6 Resources and Schedule

To cover a 9-month period:

Manpower	Man-months
Senior Staff (1 at 1/4 time)	2.25
Junior Staff (1 at 1/4 time)	2.25
Programmers (1 at full-time)	9

Equipment

Computer - Very high for development

Dedicated systems probably required to execute the RLE.

3.4.7 References

[Forsyth81] Forsyth, Daniel H., Jr., "A Remote Terminal Emulator for PR1ME Computers," School of Information and Computer Science, Technical Report GIT-ICS-81/12, Georgia Institute of Technology, Atlanta, Georgia, August 1981.

[Svobodova76] Svobodova, Liba, "Computer Performance Measurement and Evaluation Methods: Analysis and Applications," Elsevier North-Holland, Inc., 1976.

[Watkins77] Watkins, Shirley W. and Abrams, Marshall D., "Survey of Remote Terminal Emulators," National Bureau of Standards, 500-4, April 1977.

3.5 MONITORS

"Monitors" are utilized to obtain information from the target system itself during its execution.

A number of different types of monitors should be available. Some of these are:

Performance Monitors
File Utilization Monitors
Network Activity Monitors
Execution Monitors

Discussed below is the execution monitor.

3.5.1 Execution Monitors

It will be necessary to provide programmers with the proper programming tools if they are to be able to make effective use of a fully distributed processing system. The development of PRONET is an initial step in that direction, providing programming language support for the design and construction of distributed programs. The work proposed here is intended to continue this development by providing a tool for monitoring the execution of distributed programs. It will involve close interaction with other researchers participating in the Fully Distributed Processing Systems Research Program, particularly those working on the design and implementation of a distributed operating system.

3.5.2 Background

In a conventional programming environment, there are two principal purposes for monitoring the run-time behavior of a program: performance measurement and debugging. (By "monitoring" we refer to some mechanism for obtaining information about the performance of a program, external to the program itself.) Performance measurement is a relatively mundane application of monitoring in such an environment, being principally concerned with the processor time requirements of various parts of a program and requiring little or no interactive intervention by a programmer. Debugging is considerably more interesting, requiring extensive programmer interaction by its very nature. Even so, as pointed out by Plattner and Nievergelt in a recent survey [Plat81], relatively little work on debugging has been reported in the literature.

Most of the debugging tools in use today are based on concepts developed in the 60's. For instance, commonly cited papers on debugging by Evans and Darley [Evan66], Ferguson and Berner [Ferg63] and Balzer [Balz69] were all published before 1970. Many debugging tools provide access to a running program only at the machine language level. For example, a recent paper by Fairley [Fair79] reported on a tool for specifying breakpoints by assertions in assembly language programs. More sophisticated tools do allow a programmer to debug his program by interacting with it in terms of high-level language features such as variables, complex data structures and complex statement types (for example, Pierce [Pier74], Satterthwaite [Satt72] and Myers [Myer80]), but such tools are not commonly available. (It should be noted that just such a high-level view is specified for the Minimal Ada Programming Support Environment [DODR80].) Sophisticated debuggers are typically customized for a particular language, though debuggers for several languages can be built based upon a single framework, with specialized information about each language incorporated as is necessary. A debugger allowing such high-level interactions is likely to be an important part of any useful program development environment on an FDPS.

When we generalize our thinking to an FDPS from a traditional single-processor environment, the uses of monitoring become somewhat different and we must develop a new conceptual view of a major part of the monitoring task. We are, of course, still interested in performance measurement and debugging, but these tasks become quite different in this new environment. The reason for this difference is that we are now concerned with distributed programs - programs which cannot be monitored by considering a single address space on a single machine. Rather, we must now be concerned with the communication between the various parts of a program, for these interactions will play a crucial part in our monitoring task.

3.5.3 Problems to be solved

Performance measurement in an FDPS is made more complex by a number of new considerations. Use of processor time is no longer the main performance criterion. Communication costs and the overall time it takes to execute a program, which is affected by the potential for parallel execution of subtasks and by time spent waiting for messages, are equally important considerations in many situations. Further, it is much more difficult for a measurement program to monitor an entire program, since the monitored program may be

distributed arbitrarily across a network of machines. It will obviously be necessary for any monitoring program to interact with the distributed operating system of an FDPS in order to obtain the necessary information about the distribution of a program and about its communication linkage and behavior.

This need to obtain information from distributed execution sites naturally applies to debuggers as well as to performance monitors. In fact, it is a more complex problem in the case of a debugger since the debugger must somehow assist a programmer in comprehending the "state" of a program which consists of a number of processes running asynchronously on several machines. Conventional debugging tools are certainly of little use in this situation, since they are typically oriented toward monitoring the operation of what would only be a single process of a distributed program. Once again, tools which interact with the distributed operating system in order to provide information about the status of process interactions will be required. (Such tools should also have the capability to interface with more traditional monitoring tools which can be used on the individual processes.)

Just as communication should play an important part in distributed performance measurement, it should also have a crucial role in debugging distributed programs. The correctness of such programs will undoubtedly depend on the correctness of the contents and sequencing of messages transmitted between their constituent processes. Thus a distributed debugging tool must deal with communication as a major part of its job. In fact, it is conceivable that a communication monitor may be the debugger at the interprocess level, complementing traditional debuggers which operate on individual processes.

As a final difficulty, any kind of monitoring of a distributed program will potentially generate a great deal of information, which must be conveyed to a programmer in a comprehensible manner. It will presumably not be satisfactory to produce all of this information independently for each of the processes. Rather, the information must be aggregated in some manner consistent with the nature of the monitoring task being performed.

3.5.4 Proposed solutions

The network descriptors of PRONET will provide an excellent basis for the operation of distributed monitoring tools. The interconnection information these networks provide is exactly what is required by a monitor so that

it can easily recognize the structure of an entire program. Thus the implementation of a distributed performance monitor or debugger can use our PRONET work as its basis.

As was indicated in the previous section, a communication monitor will be a crucial part of any of these tools. The interconnection specifications in PRONET networks, as currently designed, provide the minimum amount of information needed by a communication monitor. That is, they provide a listing of the message paths between processes and the types of the messages which may be transmitted. The task of a monitor will be to provide a programmer with information about message transmission between processes. For performance measurement purposes, the most important information will probably involve such factors as message queue lengths and the amount of time processes spend waiting for messages. A distributed debugger, on the other hand, will be concerned with the sequencing of messages and with their contents. It will probably also be required to provide some capabilities to examine the operation of individual processes, which may be accomplished by interfacing with traditional single process debuggers.

There seem to be two useful approaches to the problem of handling the large amount of information collected by monitoring a distributed program: graphical display and automated processing of the information by the debugger. The graphical display approach would be most useful for showing the connections between processes, message queue lengths, the flow of messages, etc. Automated processing might involve such things as automatically checking for proper sequencing of messages. Extensions to the networks of PRONET to allow specification of message sequencing information would be required to make such checking possible.

3.5.5 Relationship to Other FDPS Work

Another major project in the Georgia Tech FDPS project is the development of an operating system for managing the resources of an FDPS. Preliminary work in this area has been reported in [Ensl81]. The availability of a distributed program monitoring tool should prove to be quite useful in the development and tuning of a distributed operation system (DOS). While it has been proposed that the monitor must take advantage of some operating system functions, basing the monitor on some primitive DOS capabilities while developing a full DOS should certainly be feasible. In fact, since the basic

job of a DOS is to make decisions about the distribution and scheduling of programs, evaluation of its performance will be impossible without a monitoring tool. Thus this capability needed in the course of DOS development provides an immediate motivation for the implementation of a distributed program debugger, which will be useful to all FDPS programmers.

3.5.6 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Faculty (0 m-m/year)	0
Junior Faculty (3 m-m/year)	6
Technicians (0 m-m/year)	0
Secretarial Support (1 m-m/year)	2
Student Assistants (2 students at 6 m-m/year)	24

Equipment

Computer Time	Substantial

Timing

First period of 12 months:
Port current PRONET implementation to Perqs;
design and implement communication monitor.

Last period of 12 months:
Experiment with user interfaces to debugger,
using the previously developed monitor;
interface with process-level debugger.

3.5.7 References

[Balz69] Balzer, R. M. "Exdams - Extendable Debugging and Monitoring System," AFIPS Conf. Proc., Vol. 34, (Spring 1969), pp. 567-586.

[DODR80] U.S. Department of Defense Department of Defense Requirements for Ada Programming Support Environments "STONEMAN," US Department of Defense, (February 1980).

[Ensl78] Enslow, P. H. "What is a Distributed Data Processing System?," COMPUTER, Vol. 11, No. 1, (January 1978), 13-21.

[Ensl81] Enslow, P. H. and Saponas, T. G. "Distributed and Decentralized Control in Fully Distributed Processing Systems," Technical Report GIT-ICS-81/02, School of Information and Computer Science, Georgia Institute of Technology, (February 1981).

[Evan66] Evans, T. G. and Dudley, D. L. "On-line Debugging Techniques: A Survey," AFIPS Conf. Proc., Vol. 29, (Fall 1966), pp. 37-50.

[Fair79] Fairley, R. E. "ALADDIN: Assembly Language Assertion Driven Debugging Interpreter," IEEE Trans. Software Engineering, Vol. SE-5, No. 4, (July 1979), pp. 426-428.

[Ferg63] Ferguson, H. and Berner, E. "Debugging Systems at the Source Language Level," Comm. ACM, Vol. 6, No. 8 (August 1963), pp. 430-432.

[LeBl81] LeBlanc, R. J. and Maccabe, A. B. "PRONET: Language Features for Distributed Programming," Technical Report GIT-ICS-81/03, School of Information and Computer Science, Georgia Institute of Technology, (May 1981).

[Moor81] Moore, G. L. A Distributed Compiler, M.S. Thesis, School of Information and Computer Science, Georgia Institute of Technology, (June 1981).

[Myer80] Myers, B. A. "Displaying Data Structures for Interactive Debugging," Report CSL-80-7, Xerox PARC, Palo Alto, Calif., (June 1980).

[Pier74] Pierce, R. H. "Source Language Debugging on a Small Computer," Computer Journal, Vol. 17, No. 4 (November 1974), pp. 313-317.

[Plat81] Plattner, B. and Nievergelt, J. "Monitoring Program Execution: A Survey," Computer, Vol. 14, No. 11 (November 1981), pp. 76-93.

[Satt72] Satterthwaite, E. "Debugging Tools for High-Level Languages," Software - Practice & Experience, Vol. 2, No. 3 (July/Sept. 1972), pp.197-217.

3.6 TESTBEDS FOR DISTRIBUTED SYSTEMS

3.6.1 Description

Due to the complexity of the environment, it is extremely difficult to evaluate, by analysis or simulation, the effectiveness of many algorithms and heuristics proposed for distributed systems. Evaluation is made even more difficult because many algorithms (e.g., scheduling algorithms) adapt the very environment in which they exist.

3.6.2 Background

3.6.2.1 Rationale for Testbed Development

- Realtime testing of distributed systems is a major obstacle to their development
- Use of a testbed may be the only viable alternative
- Obtaining real-time performance data may be significantly facilitated by the availability of a flexible and well-instrumented distributed testbed

3.6.2.2 Objectives in Testbed Development

- Develop a facility which will allow the advancement of distributed computing technology
- Provide the capability for rapid evaluation of architectural concepts
- Further the technology of developing high speed, high performance distributed testbed
- Standardization and integration of distributed processing technology efforts

3.6.3 Approach

One approach that has been found effective in evaluating algorithms for distributed systems is the use of a testbed which shares many of the characteristics of the environment in which the algorithms will ultimately be used. Such testbeds can be oriented toward collection of various statistics, making possible very close monitoring of the behavior of algorithms. They are particularly suited to evaluation of algorithms for concurrency control, scheduling, load distribution, distributed resource allocation, and distributed data bases.

We propose construction of such a testbed, using between 5 and 10 machines in the 0.5-1.0 MIP range, 0.5 mB main store, and Winchester disks. While the proposed machines would be considerably smaller than the ultimate systems in which the tested algorithms would be used, the machines would differ mainly in capacity, with execution speeds and communications speeds

being close to those of the ultimate machines. Details of the research proposed under this category will appear in a later version of this document.

3.6.4 Resources

A distributed system's testbed should contain at least five separate computing systems. These should be homogeneous systems, if not identical. If the target system is to be heterogeneous, then the heterogeneous components are added in addition to these.

3.7 DESIGNER WORKBENCHES

The concept of using the computer to aid in the writing of computer programs is not new. What is new is the concept of a set of automated "tools" integrated into a designers'workbench. Obviously, several different types of workbenches will be of value. The most critical need is probably in the area of distributed data base design.

3.7.1 Distributed Database Designers' Workbench

3.7.1.1 Description

The complexities of distributed data bases design far exceed the limitations of manual procedures. Evaluating the performance as well as the reliability characteristics of alternative DDB configurations requires extensive automated support.

3.7.1.2 Background

Principle work done thus far in this area has been at the University of Michigan. K.B. Irani is working in the area of "Modeling and Design of Distributed Databases and Communication Networks" and Toby J. Teorey is working with James P. Fry on "A Generalized Facility for Database Application Design."

3.7.1.3 Resources

This group is not familiar enough with this area to provide meaningful estimates.

SECTION 4
OPERATIONAL SUPPORT CAPABILITIES

4.1 INTRODUCTION

In this section, we discuss research into capabilities required for operational support for fully distributed processing systems. Such capabilities may be manifested in a production system as distinct program modules, collections of modules, algorithms used within the system, or fundamental components of the base architecture of the system. In this section we discuss file systems, command languages, load management, interprocess communication, communication protocols, and the requirements of local operating systems for support of guest, or meta, operating systems. Clearly, further operational support capabilities are required to implement a fully functioning distributed system. However, we consider that the capabilities above are fundamental to operational support of fully distributed processing systems, because they must be addressed early in the process of designing a system.

We propose a two-fold approach to the study of these capabilities. Recognizing the gains in understanding that accrue from experience in actually building systems, we suggest the construction of two testbed operating systems; one testbed will take the 'meta system' approach, and the other will be a 'native' operating system. These systems are strictly vehicles for research, however, and deviations from the task of actually designing and building them will be encouraged, in order to study issues which arise in designing their support capabilities.

The resource estimates accompanying each capability description in this section represents estimates that apply if that capability is to be studied independently of the construction of a testbed. Because the various capabilities are so inter-related, and because a testbed will allow study of more than one capability, the overall estimates for testbed construction and study of individual capabilities are less than the sum of the resources required to study each capability independently. These overall estimates are contained in the summary to this section.

4.2 DISTRIBUTED FILE AND DATA MANAGEMENT SYSTEMS

4.2.1 Description

File systems are an integral part of current operating systems and appear to be fairly well understood in a single machine context. However, in distributed surroundings, file systems are much more complicated and must support new functions, such as replication. Studying and building a comprehensive distributed transaction-based file system which supports versions, replication, concurrency control and recovery has interesting research aspects. More importantly, such a file system would be quite important in practice. Providing such features as intrinsic replication support and uniform access to data seems mandatory to attain many of the well publicized goals of distributed processing.

4.2.2 Background

Recently there has been growing interest in the problems associated with distributed file systems. There appear to be two basic varieties of file systems under analysis: server machine based and cooperating file system based. This distinction is perhaps artificial in that it may appear that the use dictates whether a system is server or cooperating based. Perhaps, another method of viewing the situation is how to evaluate the responsibility each client has in interacting with the storage system. That is, does the client have to know where (e.g., which server) the data is physically located, or does the client communicate with the local system, which then proceeds to locate and retrieve the data?

The following recent systems could be classified as server based:

WFS [Swin 79]
DFS [Stur 80]
CFS [Dion 80]
Felix [Frid 81]
Swallow [Svob 81]

The following, however, could be considered cooperating file system based:

Locus [Pope 81]
Domain [Leac 82]

Which approach is reasonable depends most likely on the hardware environment (size of local storage, distribution of hardware, etc.). Both schemes are very reasonable; we, however, will consider the cooperating file system approach because it appears to be more general, will probably be used

often in the future, and is best suited to an FDPS. Unfortunately, it is more complex. This results from the handshaking required between the file systems.

We consider the following environment. First, a client transaction requests some information (perhaps via operations on abstract objects) from a (probably local) file system A. This file system then locates the objects and the destination file system, called B, synchronizes access to them. Once the transaction completes (or aborts), the appropriate changes made to the objects are made permanent. Throughout the transaction's life, system failures have no permanent effect (with high probability [Lamp 76]) on the referenced objects.

A detailed discussion of some of the problems encountered with distributed file systems, particularly naming, is presented in Appendix A.

4.2.3 Proposed Research

The need for some means to access data in a network is obvious. However, to achieve two of the principle advantages of distributed systems, efficiency and availability, there are many problems which must be solved. Clearly, simply linking two file systems together will not suffice to achieve either efficiency or availability. For example, extending Unix path names to include the node name in the path does not address the problems of replication, transparency, or multiple file (on different nodes) commit.

The problem is to build a distributed file system supporting each of the following:

- replication
- uniform naming
- version support
- transaction based (atomic oriented)
- "standard" concurrency control (1 writer, multiple readers)

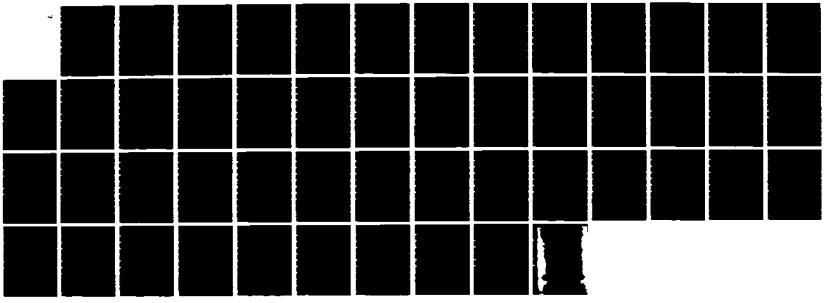
The next step is to include support for

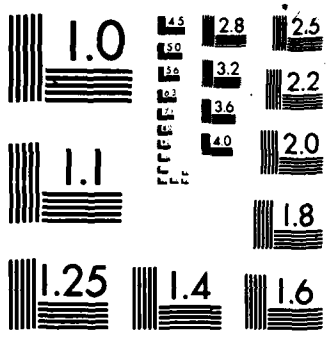
- general objects (with operations other than read and write)
- concurrency control based on specification. Thus serializability would not be the only correctness condition.

Each of the above file system aspects is discussed below. Even though security is an integral part of a file system, this research will not address this topic specifically; however, there is a research area analyzing these protection issues.

AD-A141 500 SOFTWARE SUPPORT FOR FULLY DISTRIBUTED/LOOSELY COUPLED PROCESSING SYSTEMS.. (U) GEORGIA INST OF TECH ATLANTA SCHOOL OF INFORMATION AND COMPUT.. P H ENSLOW ET AL. 2/2

UNCLASSIFIED JAN 84 GIT-ICS-82/16-VOL-1 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

4.2.3.1 Replication

To take advantage of the potential for enhanced reliability that distributed systems offer it is desirable to be able to redundantly store objects at more than one node. If the logical object is immutable, (i.e., never changes) the problem is quite simple. For mutable objects, however, updates must be coordinated so that all clients see a consistent state. There are general (and complex) solutions (e.g., [Stur 80]); however, simpler schemes such as [Leac 82] may be better.

In addition there appear to be fundamental differences in the requirements placed on replicated data. One type could be classified as amorphous, where the object (file) can be altered even during multiple node failures. The other type, primary copy, distinguishes one copy which coordinates updates. Further there is the question concerning whether all copies must be updated automatically or whether a converging approach is satisfactory.

The tradeoffs in cost (and complexity) of solving these questions are numerous. It is clear, however, that supporting general replication which is exceedingly expensive to use (because it is so general) has little merit. A more acceptable approach is to construct schemes for maintenance of replicated data which provide only weak consistency. That is, the copies of the data need not hold the same values at all times. A set of such algorithms which guarantee central consistency (in the absence of further changes) is presented in Appendix I.

4.2.3.2 Uniform Naming

In the cooperating file server approach it seems paramount to be able to hide the location of objects. Note that transparency should not, of course, be made mandatory. To have uniform naming requires that naming information may give hints as to a file's location, but cannot be absolute (an oracle). Above the unique identifier level, the system must provide user level character names. There are interesting problems in this area as well, however, they will not be considered. Many different schemes could be layered above the unique identifier level.

There are many choices when considering uniform naming. See [Leac 82] and [Shoc 78] for good discusions of these issues.

4.2.3.3 Version Support

With the advent of laser disc storage modules, versions have received

much more attention recently [Svob 81]. However, the ability to manage multiple versions is very desirable even on current hardware. There are many reasons to require version support. Two of these are software development and possibly higher concurrency. The software development reason is fairly obvious; see [Reed 78] for a discussion of the higher concurrency aspects.

4.2.3.4 Transaction Based

Users need control over what is a recoverable unit and what is a "consistent" view of changes made to files. Most systems do not provide this database approach to file storage, but it seems critical in distributed systems where failures are assumed independent. Thus, support for safe commit for multiple files on multiple nodes is required. Most commit algorithms are very expensive and some users may prefer not to be penalized for the additional safety. Thus using transctions for recovery reasons must be client controllable.

It does not seem reasonable to include support for nested transactions because of the simple nature of most operations on the files. However, if true object support is included the use of nested transactions must be reviewed. An initial specification for such a scheme is discussed in Appendix J.

4.2.3.5 'Standard' Concurrency Control

Most single machine file systems support a concurrency paradigm which is "single writers and multiple readers." This probably suffices for many applications in distributed applications too. However, in general this seems quite crude, since in any aggregate object (like a file), many updates could occur without interfering with each other. Thus "standard" concurrency seems only tentatively acceptable. This issue is elaborated in Appendix J.

4.2.3.6 General Object Support

Files can be viewed as simply instances of abstract data types with the operations of (say) open, close, read, and write. It seems quite reasonable to support the storage of general objects such as message ports, process location tables, etc., through the same basic file system mechanism. The file system then becomes a general object management system. In addition to making the system more general, higher concurrency is made possible by the system using the semantic knowledge of the operations on the objects. Further, the memory / disk data structure difference can then be ignored by the client.

Some work has been done in this area (e.g., [Poll 81]; also, Appendices H and J); however, the issues of higher concurrency have not been addressed.

4.2.3.7 Specification Based Concurrency

Most database work defines serializability as the means to define correctness of a concurrency control system. This is not always reasonable. Consider a program reading the directory of some file system. In most cases, whether the result is serializable or not is unimportant; most users do not care whether the directory list is perfect (just that it could have been in that state at some time). This is just one example, of the desire to support concurrency based on a specification which is placed with each object. The specification would define how the operations may be interleaved. Serializability could easily be specified, but many distributed applications do not require perfect serialization (e.g., naming servers and mail systems) and through the specification could weaken the correctness condition.

4.2.4 Relationship to Other FDPS Work

This support capability is driven by the requirements placed on a data storage system in a distributed system. It is operational in nature using the "best" technology available for single machine file systems and extending this model as the needs dictate for a distributed environment. It encompasses transaction research dealing with concurrency and recovery, resource replication, and version maintenance in addition to the "usual" file system problems, such as naming.

We consider the proposed research to be fundamental to distributed systems. In view of its high priority, we have begun conducting research in the area (Appendices H and J).

4.2.5 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (6 m-m/year)	12
Programmers (3 at 5 m-m/year)	30
Secretarial Support (3 m-m/year)	6

Equipment

Computer Time Substantial

Timing

First period of 12 months:
Analysis and design of data management capability.

Last period of 12 months:
Implementation & evaluation of prototype system that supports file system capabilities.

4.2.6 References

[Dion80] Dion, J., "The Cambridge File Server," Operating Systems Review, 14, 4 (October 1980), pp. 26-35.

[Frid81] Fridrich, M., Older, W., "The FELIX File Server," Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 37-44.

[Lamp76] Lampson, B., and Sturgis, H., "Crash Recovery in a Distributed Storage System," unpublished paper, CSL Xerox Parc, 1976.

[Leac82] Leach, P., Stumpf, B., Hamilton, J., Levine, P., "UIDS as Internal Names in a Distributed File System," to be presented at the Symposium on Principles of Distributed Computing, August 1982.

[Poll81] Pollack, F., Kahn, K., and Wilkinson, R., "The iMAX-432 Object Filing System," Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 137-147.

[Pope81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G., "LOCUS: A Network Transparent, High Reliability Distributed System," Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 169-177.

[Reed78] Reed, D., "Naming and Synchronization in a Decentralized Computer System," Ph.D. dissertation; Tech. Rep. TR-205, M.I.T. Lab for Computer Science, September 1978.

[Shoc78] Shoch, J., "Inter-Network Naming, Addressing, and Routing," Compcon 78, Spring 1978, pp. 72-79.

[Stur80] Sturgis, H., Mitchell, J., Israel, J., "Issues in the Design and Use of a Distributed File Server," Operating Systems Review, 14, 3 (July 1980), pp. 55-69. RE Svob81 Svobodova, L., "A Reliable Object-oriented Data Repository for a Distributed Computer," Proceedings of the Eighth Symposium on Operating Systems Principles, December 1981, pp. 47-58.

[Swin79] Swinehart, D., McDaniel, G., Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment," Proceedings of the Seventh Symposium on Operating Systems Principles, December 1979, pp. 9-17.

4.3 INTERPROCESS COMMUNICATION

The capability of a process to communicate with another on a remote node is one of the key functions a distributed processing system must support in order to attain many of the benefits claimed for distributed systems. One particular type of IPC mechanism, namely, message passing has been the focus of recent research and development because it encourages modularity and autonomy of processes. The basic functional issues of message-based IPC have been resolved, although there is and will be a continuing search for the "best" set of message passing primitives. Efficiency problems and the support for producing reliable distributed programs are among the major problems yet to be solved. The issue of remote procedure call (RPC) as a paradigm for message passing is currently a controversial area, although it is not elaborated here.

4.3.1 Background

The major advantages distributed processing systems often claim include: (1) unified access to remote resources, (2) performance improvement by parallel operations, and (3) fault tolerance through redundant resources. These advantages can be obtained by close cooperation between processes residing on separate nodes of the distributed processing system. Thus, the capability of a process to communicate with another on a remote node is one of the key functions a distributed processing system must support. Although a number of IPC mechanisms for distributed systems have been identified [ENSL79], one particular type, namely, message passing has been the focus of recent research and development. While the essential functional equivalence of message passing and other mechanisms is generally acknowledged (e.g., [LAUE78]), there have been made some arguments in favor of message passing from the sofware engineering point of view [MANN80, GENT81, STAN82]. Major advantages of message passing can be summarized in the following two points: modularity and autonomy of processes. With a message-based IPC mechanism, processes can be written to run entirely within private address spaces, disjoint from the address spaces of other processes. This modularity property enhances software understandability and maintenance. The process autonomy is derived from the generality of control flow supported by message passing mechanisms. They do not impose any hierarchy among processes. These two points, i.e., modularity and autonomy, are particularly important in fully distributed processing systems (FDPS) which require a high degree of autonomy among processes which

cooperate independently of their location. A large amount of work on IPC by message passing has been reported. A number of systems with message-based IPC facilities have been designed and/or built [AKKO74, BRIN70, CHER79, CHER81, CROW81, GIL081, HERT78, JACQ78, KAMI78, KAIN80, KRAM81, MAEK77, LI E79, RASH81, ROWE82, STIE79, TEST79, WALD72, WULF81]. High level programming languages suitable for message-based IPC are also designed and/or implemented [AMBL77, ANDR81, BRIN78, COOK80, DOD80, FELD79, HOAR78, INGA78, KESS81, LI81, LISK79, MAO80, MAY78, SILB81a, SILB81b, VAN81]. As part of the research program in Fully Distributed Processing Systems at Georgia Tech, a study on the characterization of message-based IPC facilities has been done [FUKU82], and a distributed programming language, called PRONET, has been developed [LEBL81, MACC82]. In general, the design of a message-based IPC facility must address the following basic, functional aspects of message passing: (1) how to identify the processes involved in a communication, (2) how the actual message transmission is carried out, (3) how the process synchronization can be controlled, (4) how a process can wait for and select the next message to be received, and (5) how the tools to cope with failure of communication are provided. These functional issues are essentially solved and well understood, although the way they are solved varies from one system to another.

4.3.2 Problems to be Addressed

As mentioned in the previous section, solutions to the functional problems basic to message-based IPC have been found. However, there is no consensus on the "best" set of message passing primitives whose semantics are easy to understand, efficient to implement, not error prone, yet powerful enough to allow and even encourage parallel operations. The search for such a set of message passing primitives will be continued as a main engineering issue of the distributed IPC design. There are two language-related issues in designing the functions of a distributed IPC facility. The IPC functions should be at a proper level for implementing high level programming language primitives. These functions must not be too complex (e.g., guaranteeing reliable transmission when it is not necessary) nor must they be so simple that implementation of high level language primitives for IPC using them are difficult or inefficient. On the other hand, it is desirable that the IPC functions be flexible and general enough to support multiple programming languages with different concepts of interprocess communicaton. There is very little knowledge about such interactions between the designs of the IPC func-

tions and distributed programming languages.

One of the important remaining issues of the distributed IPC design is related to efficiency problems. The slowness of message-based systems seems to be a common complaint [WULF81, CROW81]. We need to develop both general and specific techniques to minimize the overhead of message passing. Another efficiency-related problem of message-based IPC facility is how to take advantage of hardware capabilities. Some types of communication subsystems provide a capability of broadcasting a message to multiple nodes very efficiently. The IPC facility must provide the user with a concept of "single source - multiple destination" communication and implement a mechanism of effectively utilizing the broadcasting capability to deliver a message to multiple processes, some of which may reside on the same node.

The last problem we discuss here concerns the support required to produce reliable distributed programs. Since debugging a distributed program is more complex and difficult than a non-distributed program, it is highly desirable that the IPC facility recognize a user-specified "protocol" (or characteristics of conversation) among processes and do the run-time checking if the conversation conforms to the protocol. Therefore, we have to develop a protocol specification language as well as an efficient run-time checking mechanism.

4.3.3 Proposed Solutions or Initial Approaches

Concerning the search for the "best" set of functions to be provided by the IPC facility, we have to gain more experience of the performance of various message-based IPC facilities as well as experience in building various types of distributed programs. With respect to the efficient implementation of a message-based IPC facility, the recent paper by Spector [SPEC82] shows an interesting and encouraging approach. Spector's experiment shows that simple remote operations can be executed quickly (about 150 microseconds including the transmission time on Xerox Alto computers using the 2.94 megabit Ethernet, which is about two orders of magnitude faster than could be expected if they were implemented in a conventional way). This improvement in performance is made possible by the specialization of the communication interface, the use of simplified protocol, and the direct implementation in microcode. An appropriate initial approach to the problem of "protocol" specification and run-time checking of conversations among processes can be found in [LIVE80].

A language, called the "Task Graph Language", allows the specification of connectivity among processes, message sequencing, concurrency, and mutual exclusion. The constraints specified by a task graph are then enforced at run time. In the research we propose, the fundamental issue to be examined is that of deciding which characteristics of interprocess communication mechanisms are best suited to fully distributed processing systems. For this purpose, we propose that the testbeds be constructed with a view to evaluation of different interprocess communication protocols. This is not an easy task, since interprocess communication is usually an integral component of an operating system. Once such a testbed has been constructed, however, it can be used to evaluate numerous protocols by experimentation and analysis of difficulty of use and overhead incurred.

4.3.4 Relationship to Other FDPS Work and SSC's

Since distributed IPC is one of the key capabilities in distributed processing systems, its design requires close cooperation with the design of other operational support capabilities, particularly distributed access control, distributed monitoring, distributed file management, distributed recovery management, and communication protocols. The functions provided by a distributed IPC facility also affect the design of some software support tools, particularly program design languages, distributed programming languages, and interactive monitors.

4.3.5 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (6 m-m/year)	12
Programmers (3 at 5 m-m/year)	30
Secretarial Support (3 m-m/year)	6

Equipment	
Computer Time	Several work stations

Timing

First period of 12 months:
Build simulator for distributed IPC.

Last period of 12 months:
Conduct experiments in distributed IPC.

4.3.6 References

[AKKO74] Akkoyunlu, E., Bernstein, A., and Schantz, R. Interprocess communication facilities for network operating systems. IEEE Computer 7, 6 (Jun. 1974) 46-55.

[AMBL77] Ambler, A.L., Good, D.I., Browne, J.C., Burger, W.F., Cohen, R.M., Hoch, C.G., and Wells, R.E. GYPSY: a language for specification and implementation of verifiable programs. Proc. ACM Conf. on Language Design for Reliable Software, ACM SIGPLAN Notices 12, 3 (Mar. 1977) 1-10.

[ANDR81] Andrews, G.R. Synchronizing Resources. ACM Trans. Programm. Lang. Syst. 3, 4 (Oct. 1981) 405-430.

[BRIN70] Brinch Hansen, P. The nucleus of a multiprogramming system. Comm. ACM 13, 4 (Apr. 1970) 238-241, 250.

[BRIN78] Brinch Hansen, P. Distributed processes: a concurrent programming concept. Comm. ACM 21, 11 (Nov. 1978) 934-941.

[CHER79] Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R. Thoth, a portable real-time operating system. Comm. ACM 22, 2 (Feb. 1979) 105-115.

[CHER81] Cheriton, D.R. The design of a distributed kernel. Proc. ACM Annu. Conf. (Nov. 1981) 46-52.

[COOK80] Cook, R.P. *MOD -- a language for distributed programming. IEEE Trans. Softw. Eng. SE-6, 6 (Nov. 1980) 563-571.

[CROW81] Crowley, C. The design and implementation of a new UNIX kernel. [ul AFIPS Conf. Proc. Vol.50 1981 Natl. Computer Conf. (May 1981) 265-271.

[DOD] DOD (US Department of Defense). Reference Manual for the Ada Programming Language - proposed standard document. July 1980.

[ENSL79] Enslow, P.H., and Gordon, R.L.(Eds.) Interprocess communication in highly distributed systems - a workshop report. Tech. Rep. GIT-ICS-79/11, School of Info. & Comput. Sci., Georgia Institute of Technology (Dec. 1979).

[FELD79] Feldman, J.A. High level programming for distributed computing. Comm. ACM 22, 6 (Jun. 1979) 353-368.

[FUKU82] Fukuoka, H. Interprocess communication facilities for distributed systems: a taxonomy and a survey. School of Info. and Comput. Sci., Georgia Institute of Technology (submitted to the ACM Computing Surveys).

[GENT81] Gentleman, W.M. Message passing between sequential processes: the reply primitive and the administrator concept. Software-Practice and Experience 11, 5 (May 1981) 435-466.

[GIL081] Giloi, W.K. and Behr, P. An IPC protocol and its hardware realization for a high-speed distributed multicomputer system. .Proc. 8th Annu. Symp. on Computer Architecture (May 1981) 481-493.

[HERT78] Hertweck, F., Raubold, E., and Vogt, F. X25 based process-process communication. Computer Networks 2 (1978) 250-270.

[HOAR78] Hoare, C.A.R. Communicating sequential processes. Comm. ACM 21, 8 (Aug. 1978) 666-677.

[INGA78] Ingalls, D. The Smalltalk-76 programming system: design and implementation. Conf. Rec. 5th Annu. ACM Symp. on Principles of Programming Languages (Jan. 1978), 11-16.

[JACQ78] Jacquemart, Y.A. Network interprocess communication in an X25 environment. Computer Networks 2 (1978) 227-235.

[KAIN80] Kain, R.Y., and Franta, W.R. Interprocess communication schemes supporting system reconfiguration. Proc. COMPSAC 80 (Oct. 1980), 365-371.

[KAMI78] Kamibayashi, N., Ata, Y., Akatsuka, H., and Aiso, H. Distributed processing oriented interprocess communication facility for KOCOS. Proc. 3rd USA-Japan Computer Conf. (Oct. 1978) 80-85.

[KESS81] Kessels, J.L.W. The Soma: a programming construct for distributed processing. IEEE Trans. Softw. Eng. SE-7, 5 (Sep. 1981) 502-509.

[KRAM81] Kramer, J., Magee, J., and Sloman, M. Intertask communication primitives for distributed computer control systems. Proc. 2nd Int. Symp on Operating Systems (Apr. 1981) 404-411.

[LAUE78] Lauer, H.C., and Needham, R.M. On the duality of operating system structures. Proc. 2nd Int. Symp. on Operating Systems, IRIA (Oct. 1978), reprinted in ACM Oper. Syst. Rev. 13, 2 (Apr. 1979) 3-19.

[LEBL81] LeBlanc, R.J., and Maccabe, A.B. PRONET: language features for distributed programming. GIT-ICS-81/03, School of Info. and Comput. Sci., Georgia Institute of Technology (May 1981).

[LI] Li, C.M., and Liu, M.T. DISLANG: a distributed programming language/system. Proc. 2nd Int. Symp on Operating Systems (Apr. 1981) 162-172.

[LISK79] Liskov, B. Primitives for distributed computing. Conf. Rec. 7th Annu. ACM Symp. on Principles of Programming Languages (Dec. 1979) 33-42.

[LIVE79] Livesey, J. Inter-process communication and naming in the Mininet system. Digest of Papers COMPCON Spring 79 (Feb. 1979) 222-229.

[LIVE80] Livesey, J., and Manning, E. Protection and synchronization in a message-switched system. Proc. Working Conf. on Operating Systems Engineering, IBM Japan (Oct. 1980) 373-417.

[MACC82] Maccabe, A.B., and LeBlanc, R.J. The design of a programming language based on communication networks. School of Info. and Comput. Sci., Georgia Institute of Technology (submitted to the 3rd Int. Conf. on Distributed Computing Systems).

[MAEK77] Maekawa, M. Interprocess communications in a highly diversified distributed system. IFIP Information Processing 77 (1977) 149-154.

[MANN80] Manning, E., Livesey, N.J., and Tokuda, H. Interprocess communication in distributed systems : one view. IFIP Information Processing 80 (Oct. 1980) 513-520.

[MAO] Mao, T.W., and Yeh, R.T. Communication port: a language concept for concurrent programming. IEEE Trans. Softw. Eng. SE-6, 2 (Mar. 1980) 194-204.

[MAY] May, M.D., Taylor, R.J.B., and Whitby-Strevens, C. EPL - an experimental language for distributed computing. Proc. Trends and Applications: Distributed Processing. (May 1978) 69-71.

[RASH81] Rashid, R.F., and Robertson, G.G. Accent: a communication oriented network operating system kernel. Proc. 8th Symp. on Operating Systems Principles (Dec. 1981) 64-75.

[ROWE82] Rowe, L.A., and Birman, K.P. A local network based on the UNIX operating system. IEEE Trans. Softw. Eng. SE-8, 2 (Mar. 1982) 137-146.

[SILB81a] Silberschatz, A. Port directed communication. Computer J. 24, 1 (Feb. 1981) 78-82.

[SILB81b] Silberschatz, A. A note on the distributed program component Cell. ACM SIGPLAN Notices 16, 7 (Jul. 1981) 89-96.

[SPEC82] Spector, A.Z. Performing remote operations efficiently on a local computer network. Comm. ACM 25, 4 (Apr. 1982) 246-260.

[STAN82] Stankovic, J.A. Software communication mechanisms: procedure calls versus messages. IEEE Computer 15, 4 (Apr. 1982) 19-25.

[STIE79] Stiegler, H.G. A unified interface for process communication. Proc. 1st Int. Conf. on Distributed Computing Systems (Oct. 1979), 419-429.

[TEST79] Test, J.A. An interprocess communication scheme for the support of cooperating process networks. Proc. 1st Int. Conf. on Distributed Computing Systems (Oct. 1979), 405-411.

[VAN] van den Bos, J., Plasmeijer, R., and Stroet, J. Process communication based on input specifications. ACM Trans. Programm. Lang. Syst. 3, 3 (Jul. 1981), 224-250.

[WALD72] Walden, D.C. A system for interprocess communication in a resource sharing computer network. Comm. ACM 15,4 (Apr. 1972) 221-230.

[WULF81] Wulf, W.A., Levin, R., and Harbison, S.P. HYDRA/C.mmp: An Experimental Computer System. McGraw-Hill (1981) 91-104.

4.4 COMMAND LANGUAGES

4.4.1 Description

The command language used in a fully distributed processing system is a critical component, as it represents the FDPS to the user. Similar to the relationship between a programming language and a compiler or between a command language and an operating system, a distributed command language may be considered as being implemented by an FDPS. The term "user" is meant to be general, describing whatever is at the end-points of the FDPS. Examples of users include application programs as well as people at various levels of use such as application users, application designers and implementors, and system implementors. A command language includes both the commands, which request action of an underlying system, and the responses which are returned by the system, indicating the status of the requested action. The command language may be seen by a user as a programming interface, or as a series of messages exchanged on a terminal. Commands may either specify requirements, allowing the underlying system to determine how these are to be met, or may be procedural, specifying how the action is to be carried out.

4.4.2 Background

Historically, command languages have been developed in order to provide the capabilities of operating systems to users. The design has typically been structured around these operating system capabilities, giving the user a somewhat abstract view of the operating system. As this view is defined by the operating system rather than user requirements, the user is faced with a "semantic gap" which is filled by becoming familiar with aspects of the machine which are not related to the user's task. This can be seen in the proliferation of unique command languages available for various machines, which require a user to learn more details of a particular machine than should be necessary to get the task completed.

Command languages for computer networks have followed a similar trend. For example, level models such as the OSI model are developed in a bottom-up order, with attention being paid last to the top levels, where user command languages are defined [Hertweck80]. Needed is more dialogue between network and command language designers, such as the relation which exists between compiler writers and language designers. This should take place before any standardization in order to avoid standardizing outdated techniques, such as batch punched-card workstations.

A distributed command language for an FDPS would be classified as a "network operating system command and response language", or NOSCRL. Standardization activities are taking place by several groups, ANSI [FramptonMellor-Schlegel80], British Computer Society [Newman80b], and CODASYL [Harris80], though little is being done in the areas of research [Hertweck80].

4.4.2.1 Options for Common Command Languages

There exists a wide variety of command languages and design philosophies for them; i.e., some consider them to be simply a job control language, some consider them to be a collection of tools and some insist that a command language should be as powerful as any programming language. How should a command language or possibly multiple command languages fit into a fully distributed processing system? The alternatives for incorporating command languages into distributed systems (but not necessarily an FDPS because some of the options do not meet the defined characteristics of an FDPS,) are:

1. Allow only one command language in the entire network.
2. Allow for one common command language that all nodes in the network must understand. A given node may provide other command languages but it must first provide a 1-1 translation between its local command language and the common network command language. Users may access the network from any node using either the common network command language or one of the available local command languages.
3. Do not provide a common network command language but allow any command language in the network provided that there are translators written from that command language to every other command language in the network.
4. Do not provide for a complete common network command language, but define a subset of network commands with which any command language in the network must have 1-1 correspondence. For example there should be a one to one correspondence between file copy commands and mail and message sending facilities.
5. Do not provide for a network command language nor insist that translators be written between different command langauges.

4.4.2.2 Load-Based Command Languages

Since the distributed command language contains information from the entire set of concurrent processes making up a user's job, it can be used to convey much more information to the local operating system than simply which processes are to run concurrently, and which processes communicate with whom.

We simply list here some of the information which can potentially be added to command language statements in order to supply extra information to

the operating system in order to aid in scheduling and work distribution decisions:

Inter-process Communication Mechanism
Volume of Inter-process Communication
Command Location
Program Size
Concurrency and Sequentiality

4.4.3 Problems to be Addressed

The definition and implementation of a distributed command language has many problems associated with it. Some are general cases of problems faced by an operating system command language, such as specification of resource requirements, while others are unique to the environment of an FDPS. Taking the view that the distributed command language should be user-oriented, the problems described here are mainly those from a user's point of view.

Visibility of Network. A friendly network does not interfere with users, but provides services to meet users' requirements. The network should logically be considered as a passive communications medium, and should be as transparent to the user as possible.

Richness vs. Simplicity. A command language which expresses all the capabilities of a system is powerful, but most likely difficult to use. A simple command language, while easy to use, may not have the ability to handle complex requests.

Tailored vs. Common Language. As mentioned previously, a standard NOSCRL is being worked on by several groups, where ideally a user would need only the common language to specify requirements, allowing portability among processors in a network or between networks. On the other hand, user requirements vary, for example between the application package user and the application designer. Therefore a single standard communications language may be too complex to adequately serve the range of users, requiring either different languages or different levels of access within the language.

Location of Command Language. A command language can be implemented either within one or more existing programming languages, or separately as a language by itself. The main argument for inclusion within programming languages is that users need to know only the programming language -- but many users don't want to know any programming language. The question also arises as to which programming languages should have the command language embedded.

A command language may be considered as a means of specifying the environment in which a program is to run, in which case it would be used separately from application programs.

Command Syntax. The syntax of a command language has an important effect on its usability. One main difference in style is the use of either keyword parameters or positional parameters. Keyword parameters are self-documenting, but take longer to enter. Use of menus is similar. Some approaches allow either, such as procedure calls in Ada.

Human Factors. Under this category fall the various aspects of man-machine interfaces which make the distributed command language easier to use from a human user's standpoint, as well as within applications which are prepared by humans. New terminal types, such as those with intelligence or graphics capabilities, open new possibilities for making machines easier to use.

Compilation vs. Translation. A command language could either be translated, where commands are acted on immediately upon entry, or compiled, which allows the system to have all the user's requests available to perhaps make better decisions on such things as resource allocation. Also, in what form should command procedures be kept -- compiled or in the original source? An extensive discussion of this topic is presented in Appendix B.

"User" Interfaces. It is becoming fairly widely accepted that there are several different "levels" of user interfaces required in a system. (For a recent discussion see Roger W. Ehrich and H. Rex Hartson, "On Effective Software Development Methodology," CACM, Vol. 25, No.5, May, 1982, pp.350-351.) It is not yet clear exactly what user levels of control will be required; however, the need for at least the following appears obvious for distributed systems:

- System Programmer
- System Manager
- Maintenance Personnel
  - Hardware
  - Software
- Applications Systems Programmers
- "Occasional" Programmers
- Non-Programming Users

4.4.4 Proposed Solutions

Several approaches to the definition of a distributed command language will be described. These are not disjoint, but describe major concepts which address the problems.

Translation Between Systems [KraylUngerWeller75]. A common command language (NOSCRL) is defined, but used only for communication between different operating systems. The user sees a system with which he is familiar, and enters commands in its format. The commands are translated into the command language(s) of the system(s) on which the user's task is to be performed, in two stages. First, the commands are translated into the common, or intermediate, language by the system whose host language was used. Then, the commands in the common language are translated on the destination system(s) into the local command language, and executed. Each system requires only two "half-translators", to translate between the local command language and the common command language.

Translation Into Systems. [Dakin75], [Newman75], [Newman80a]. Commands from users are entered in a common command language (NOSCRL), and then are translated into the command language of the target system. They may be translated on the local system, or on the target system where they are executed.

Message-Oriented Model [LauerNeedham79]. A system built on the message-oriented model is comprised of processes which pass messages among themselves, as opposed to the procedure-oriented model, in which processes move between contexts. The message-oriented model is closer in structure to a distributed system than the procedure-oriented model, though the latter can be provided through "communicating variables" [Hertweck80]. The message-oriented model allows the general definition of a user as a process which autonomously communicates, and may be a program or person at a terminal. Commands from a user are messages directed at processes which manage resources. Such a command language designed around message protocols could have a single specification, regardless of whether a command originated from a terminal or a program.

Virtual Protocols have been used successfully to interface system components, especially in the area of communication systems. Based on a virtual model, such as a file or terminal, commands are defined to carry out its operation.

Abstract Machines [Unger80], [Kugler80], [Hopper80]. The objective of this approach is to "define a communication interface between a computer system and all its users, which enables every user to solve his problems on a semantic level appropriate to his problems (and not necessarily to those of the computer system)." [Unger80]. Looking at a computing system from a user's perspective, the user sees an operating system command language, which is implemented by the operating system. A taxonomy of universal operating system facilities can be established, and a given operating system can be considered as implementing a subset of these facilities. A user's requirements do not depend on which machine is to be used. However, the operating system command language varies considerably from one machine to another, forcing the user to understand how his requirements can be met through the subset of the universal operating system facilities available through the unique command language on a particular system.

The approach taken by use of an abstract machine (AM) is to define the operating system facilities in a consistent manner as a "basic abstract machine" (BAM), which is implemented in a layer above the operating system. The facilities of the BAM are then the basis for the definition of several AMs, each tailored for a particular user. The AMs are portable as the BAM is a standard, regardless of which machine it is implemented on. (though presumably all machines do not provide all resources).

In a network environment, this concept is applied to the definition of "virtual network machines", which provide resources of one or more real machines to users. This approach allows portability of a user's tailored command language among network machines, but does not require all users to follow the same command language as does a standard NOSCRL.

4.4.5 Initial Approaches

Research into this capability can be conducted analytically, or experimentally in conjunction with either or both of the two testbeds. Problems encountered in the guest system testbed will be more driven by the state of current technology, while the native testbed will offer an opportunity to work in a less constrained environment.

The figures below are for a pilot project to investigate the requirements of distributed command languages analytically at first, with a 'simulated' environment to be constructed in the second year to evaluate

results. More extensive research can be done on this topic on construction with testbed construction.

4.4.6 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (6 m-m/year)	12
Programmers (3 at 5 m-m/year)	30
Secretarial Support (3 m-m/year)	6

Equipment

Computer Time	Substantial

Timing

First period of 12 months:
Analysis and design of command language capabilities.

Last period of 12 months:
Construction and evaluation of simulator for distributed command languages.

4.4.7 References

[Dakin75] Dakin, R.J., "A General Control Interface for Satellite Systems", in Command Language, North Holland, 1975, pp. 281-290

[FramptonMellorSchlegel80] Frampton, L.C., Mellor, S., and Schlegel, C.T., "A Standard Operating System Command and Response Language", in Command Language Directions, North Holland, 1980, pp. 83-93

[Harris80] Harris, T.J., "The CODASYL COSCL Journal of Development", in Command Language Directions, North Holland, 1980, pp. 119-125

[Hertweck80] Hertweck, F., "Computer Networks: Recent Trends and Issues Related to Command Languages", in Command Language Directions, North Holland, 1980, pp. 369-383

[Hopper80] Hopper, K., "The KIWINET/NICOLA Approach: Implementation in a Heterogeneous Network with at Least One Node", in Command Language Directions, North Holland, 1980, pp. 389-401

[KraylUngerWeller75] Krayl, H., Unger, C., and Weller, T., "Portability of Job Control Language Programs", in Command Language, North Holland, 1975, pp. 293-302

[Kugler80] Kugler, H.J., "The KIWINET/NICOLA Approach: Tools for the Construction of User Interfaces", in Command Language Directions, North Holland, 1980, pp. 299-316

[LauerNeedham79] Lauer, H.C., and Needham, R.M., "On the Duality of Operating System Structures", in Operating Systems: Theory and Practice, North Holland, 1979, pp. 371-374.

[Newman75] Newman, I. A., "Machine Specific Facilities in a Machine Independent Command Language", in Command Language, North Holland, 1975, pp. 91-104

[Newman80a] Newman, I. A., "Developments in the UNIQUE Machine-Independent Command Language", in Command Language Directions, North Holland, 1980, pp. 65-78

[Newman80b] Newman, I. A., "A Model System and a Model of User Interactions: the BCS JCL WG View", in Command Language Directions, North Holland, 1980, pp. 101-113

[Rayner80] Rayner, D., "User Interfaces in Open Data Communication Networks", in Command Language Directions, North Holland, 1980, pp. 405-416

[Unger80] Unger, C., "The KIWINET/NICOLA Approach: Concepts", in Command Language Directions, North Holland, 1980, pp. 281-297

4.5 LOAD MANAGEMENT

4.5.1 Local Scheduling

Local scheduling involves deciding when to assign resources (e.g., physical memory, processor) to eligible processes so that goals of response time and throughput are met. This section is concerned only with the resources available at the local site. The section on work distribution discusses other alternatives.

4.5.1.1 Background

There are two basic types of scheduling : deterministic and probabilistic. Deterministic scheduling is only possible when the processing time for each process or task is known beforehand. A significant body of work has been done in this area. Coffman and Denning [Coffman & Denning, 73] provide a good introduction. Much of the work has been done for multiple processors, and thus may be applicable to distributed systems where processing times are known a **priori**. However, this is usually not the case. When processing times are not known beforehand, probabilistic scheduling is used. Probabilistic scheduling has many heuristic characteristics. In "classical" systems, it involves techniques such as round-robin and priority queueing disciplines. Several schemes have been proposed for the special environment of distributed systems. One such is the concept of **coscheduling** in Medusa [Ousterhaut, et al., 80]. A task force (i.e., a set of cooperating processes) is said to be coscheduled if all of its runnable processes are simultaneously scheduled for execution on their respective processors. Thus, most interprocess communication can proceed immediately, since the communicating processes are both currently running. (Note that this assumes relatively short delays for communications.)

Another approach is the **wave scheduling** technique used in MICROS [vanTilborg & Wittie, 81]. This involves structuring processes into trees, where each level of the tree consists of managers for the level below. Scheduling is done hierarchically, with each level of managers scheduling the level below. A "wave" propogates down the tree with each high-level scheduling decision.

4.5.1.2 Problems and Initial Approaches

Neither of the approaches was specifically designed for distributed systems. To evaluate their utility in this environment, a possible direction

for research would be to monitor processes' requests for activation, seeking relationships between groups of processes. Such relationships would then be used to develop and evaluate new scheduling algorithms.

4.5.2 Work Distribution

4.5.2.1 Description

Work distribution for FDPS's involves assignment of resources (e.g., files, devices, processors) so that goals of system utilization, response time, and throughput are met. This problem has long been studied in the context of centralized systems. In the case of an FDPS, however, the problems introduced by the nature of the FDPS environment make the problem much more difficult (e.g., time delays in communication, possible failures, autonomy, security, etc.). Other problems closely related to workload distribution are: process and file migration, node autonomy, decentralized control (decision making).

4.5.2.2 Background

In order to make the best use of the multiplicity of resources available in an FDPS, there must be some coherent policy set forth and enforced. In a situation where each site has all the resources it will ever need, work distribution may not be necessary. However, if this is the case, then it is most likely that each site will not always be using all of its local resources. Some form of work distribution is necessary in order to utilize these idle resources. Previous work in this area can be broken down into two categories: placement and assignment [Jones & Schwarz, 80] [Sharp, 82]. The placement problem involves the physical placement of resources (i.e., files) in the network. Allocation of processes to processors constitutes the assignment problem. The placement problem has received the most attention [Buckles & Hardin, 79] [Casey, 72] [Chang & Liu, 79] [Chen & Akoka, 80] [Chu, 69] [Chu, 73] [Irani & Khabbaz, 79] [Levin & Morgan, 75]. The approaches range from optimal graph theoretic solutions (of limited applicability) to heuristic algorithms and simulation. The assignment problem has received somewhat less attention. Most of the work [Rao, et al., 79] [Stone, 77] [Stone, 78] [Stone & Bokhari, 78] has been of a graph theoretic nature, and the algorithms quickly become computationally intractable when extended to even a moderate number of processors. The general assignment problem has been shown to be NP-complete [Kratzer & Hammerstrom, 80]. Casey and Shelness [Casey & Shelness, 77] have proposed a heuristic that shows promise.

Simulations [Sharp, 82] have also been done.

4.5.2.3 Problems

A problem requiring study is transparent process migration; i.e., how to relocate a process such that the process is unaware that it has been moved. There is a large amount of state information associated with an active process that must be maintained consistently during the transport. Also, if the process is communicating with other processes, the time spent in migration can cause other processes to timeout (considering the migrating process to have failed) unless precautions are taken. Possibly, a more profitable approach might be to consider migrating "transactions" as units of work, rather than entire processes.

A second, more fundamental problem is that of the decision apparatus. The decision to distribute load can be made by a logically centralized "workload controller", or by one of a number of nodes if a decentralized scheme is used. Equally important is what information is used in making the decision, and how that information is maintained. Many issues in this area are discussed by Jensen in [Jensen81].

4.5.3 Initial Approaches

Perhaps the most important things to know when designing a workload distribution mechanism (or, in deciding if one is indeed necessary) are the characteristics of the workload expected for the system in question. A distribution scheme that works well in an interactive software development environment may be completely inadequate for a real-time command-and-control system. Also, a scheme that can handle both environments may be too slow to be useful to either. Thus, the workload characteristics, together with the purposes and goals of the system at hand, will greatly impact the design of the workload distributor.

Modeling and simulation can be used to achieve this characterization, but the best method is probably direct measurements from an existing system that implements the same (or similar) functions. Extrapolation can then be made to include any enhanced functionality to be provided by the new system (here, modelling and simulation are necessary).

The computational intractability of the distribution problem requires the use of heuristics in any practical system. The only way to evaluate these heuristic algorithms is through the use of simulations or experiments. Some

of the more promising work [Casey & Shelness, 77] has used this approach.

Simulation as a technique for evaluating algorithms in distributed systems is limited in the extent to which it can capture the volatility and dynamic nature of the system, and the extent that it can detect unexpected transient effects which might be vital to a working system. Therefore, we propose incorporating research into load management into the construction of a testbed, to take place after the initial testbed is constructed. If no testbed is to be constructed, we propose that a hardware configuration, similar to that of an FDPS, be constructed. Upon this hardware, a truly distributed simulation system can be built. Such a system will capture the effects of line transmission delays and internal queueing in the nodes.

4.5.4 Relationship to Other FDPS Work

Load management is part of the issue of general resource management. As such, it is associated closely with command languages. A relationship is also seen with data management, since the information upon which resource allocation decisions are made is distributed in nature.

4.5.5 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (6 m-m/year)	12
Programmers (3 at 5 m-m/year)	30
Secretarial Support (3 m-m/year)	6

Equipment

Computer Time	A Loosely-coupled multiple processor testbed

Timing

First period of 12 months:
Build distributed simulator.

Last period of 12 months:
Conduct experiments in distributed load managment.

4.5.6 References

[Buckles79] Buckles, B. P., and D. M. Hardin, "Partitioning and allocation of logical resources in a distributed computing environment," General Research Corp., 79.

[Casey72] Casey, R. G., "Allocation of copies of a file in an information network," AFIPS Conference Proceedings, 41, 1, 72.

[Casey77] Casey, L. and Shelness, N., "A domain structure for distributed computer systems," Proc. Sixth ACM Symp. on Operating System Principles, November 77.

[Chang79] Chang, S. R., and Liu, C. N., "Modeling and design of distrubuted information systems," Advances in Computers, New York: Academic Press, 79.

[Chen80] Chen, P. P., and J. Akoka, "Optimal design of distributed information systems," IEEE Trans. on Computers, C-29, September 80.

[Chu69] Chu, W. W., "Optimal file allocation in a multi-computer information system," IEEE Trans. on Computers, C-18, 69.

[Chu73] Chu, W. W., "Optimal file allocation in a computer network," In Abramson and Kuo (Eds.), Computer-Communication Networks, Englewood Cliffs: Prentice-Hall, 73.

[Coffman73] Coffman, E. G., and P. J. Denning, Operating Systems Theory, Prentice-Hall: Englewood Cliffs, 73.

[Irani79] Irani, K. B., and N. G. Khabbaz, "A model for a combined communication network design nad file allocation for distributed databases," Proc. Second Int. Conf. on Dist. Computing Systems, april, 81.

[Jensen81] Jensen, E.D., "Distributed Control," in Distributed Systems - Architecture and Implementation, Springer Verlag, Vol.105, 1981.

[Jones80] Jones and Schwarz, "Experience using multiprocessor systems - a status report," ACM Computing Surveys, June 80.

[Kratzer80] Kratzer, A., and D. Hammerstrom, "A study of load levelling," Proc. COMPCON Fall 80.

[Levin75] Levin, D. K., and H. L. Morgan, "Optimizing distributed databases - a framework for research," AFIPS Conference Proceedings, 44, 75.

[Ousterhaut80] Ousterhaut, J. K., D. A. Scelza, and P. S. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," CACM, Vol. 23, No. 2, February 80.

[Rao79] Rao, G. S., H. S. Stone, and T. C. Hu, "Assignment of tasks in a distributed processor system with limited memory," IEEE Trans. on Computers, C-28, April 79.

[Sharp82] Sharp, D., "Work distribution in a fully distributed processing system," Ph.D. Thesis, Georgia Institute of Technology, GIT-ICS-82/01, 82.

[Stone77] Stone, H. S., "Multiprocessor scheduling with the aid of network flow algorithms," IEEE Trans. on Software Engineering, SE-3, January 77.

[Stone78] Stone, H. S., "Critical load factors in two-processor distributed

systems," IEEE Trans. on Software Engineering, SE-4, May 78.

[Stone78] Stone, H. S., and S. H. Bokhari, "Control of distributed processes," Computer, July 78.

[vanTilborg81] van Tilborg, A. M., and L. D. Wittie, "Wave Scheduling: Distributed Allocation of Task Forces in Network Computers," Proceedings of the Second International Conference in Distributed Computing Systems, IEEE, April, 81.

4.6 META SYSTEMS

A distributed operating system (DOS) is a set of software capabilities which manage the resources of a distributed processing system. The DOS requires support in the form of local operating systems on the various nodes in the system. The implementation of such an operating system can proceed in two ways: the local operating systems which will provide the support for the DOS may be designed from scratch, with the functionality required for the DOS in mind, or the DOS may be implemented as a layer above already existing operating systems.

A DOS implemented with the latter approach is called a guest system or meta-system. The local operating systems used by guest systems were not necessarily designed to support anything other than local access of resources.

This section describes some of the problems encountered by the guest system approach, particularly those systems based on heterogeneous host systems. Also some approaches taken to solve these problems are discussed.

4.6.1 Background

The agency responsible for providing FDPS users with services is the distributed operating system (DOS). A DOS differs from a traditional operating system in that its fundamental concern is not with the sharing or multiplexing of resources ([PEEB80], [KIMB76], [FORS80], and [WATS80]). Rather, the DOS makes services available to users, and establishes global policies concerning the use of these services. For example, if there is a class of services providing essentially the same function, the DOS decides which of this set a user is allocated.

The DOS is also responsible for locating services for the user. Users of the system should be able to ask for services by logical names.

Because a FDPS is composed of several processors, programs written for these systems may take advantage of the parallelism available. Such programs would be composed of modules which communicated by passing messages. The DOS is responsible for providing inter-process communication (IPC). IPC should appear the same, regardless of whether the processes involved are using the same processor or on different processors.

The DOS is also responsible for distributed process management. This involves the creation and destruction of processes at a global level. It may

also be necessary for the DOS to be able to block processes.

The DOS must provide for the protection of resources from incorrect or unauthorized usage, similar to the service provided by traditional operating systems. The service provided by the DOS may be more complicated, so that a user may not have access to a service that resides on a particular host, but may be allowed to use a similar service on a different host. For example, if a host on the system is being used for developmental purposes, access to its local resources may be restricted.

In order to provide these functions, the DOS must rely on local operating systems present on each of the host machines in the system. It is these local operating systems which will provide the traditional operating system functions (memory management, scheduling, and so forth) and manage the local resources of a machine. In order for the FDPS user to make use of the services, the DOS must request the service from one of the local operating systems.

4.6.2 Guest Systems

The local operating systems for the host machines may be designed from scratch with the express purpose of supporting the DOS. The DOS is then implemented as part of these systems. This is called the base level approach [THOM78]. This approach allows the functions of the DOS to be considered at the local operating system level. The resulting system can be very efficient, since the host operating systems and the DOS are designed to mesh together into a cohesive system. Indeed, the prime advantage of this approach is the possibility of integrating the functions of the DOS and host operating systems to some degree.

The main handicap to this approach is the cost of development. Not only must a code for distributed functions be written, but the code necessary to handle the traditional operating system services must also be written. The meta-system [THOM78] or guest system approach avoids this drawback by using existing operating systems as the host systems. Using the meta-system approach, the DOS becomes a layer of software that runs on top of the local operating systems. It is essentially an application program which transforms requests for distributed services into the appropriate requests for services that the local operating system provides.

One advantage of the guest system approach was already mentioned. Indeed, much early research into distributed systems assumed the guest system approach for this very reason ([KIMB76], [FORS77], [MILL77]). However, another advantage for taking this approach is that most existing systems have considerable investment in application software which would become useless if the underlying operating system were thrown away. If the purpose of designing the distributed system is to allow users access to a wide range of such software, then the guest system approach would seem more advantageous.

The NSW, for example, was designed to allow users access to the wide range of services which exist on the various hosts in the system ([GELL77] and [MILL77]. The system was designed to run on Tenex, Multics, and OS/360 systems. NSW was intended to allow users at various locations in the system to share software development tools.

ADAPT [PEEB80] is a guest system which is intended to run on VAX/VMS. ADAPT is an object model system. ADAPT sees resources as typed objects that can be operated upon by a limited number of functions. ADAPT attempts to use existing software as much as possible, so it does not take the object model to its full extreme, using relatively large structures, such as files, as the limits of granularity.

Desperanto is a guest system designed to run on a variety of systems [MAMR82a]. The system views distributed software as a set of modules. A module is a set of data objects and a set of functions which can manipulate the objects.

4.6.3 Research Problems

The basic problem faced by a guest DOS is the translation of local services into FDPS services. The most common solution to this problem is to require that each host in the system support some sort of monitoring process which is responsible for requesting services from the host system ([MAMR82a], [PEEB80], [FORS78]). This monitoring process is the interface between the FDPS user and the underlying host system. Since the monitoring process is running at what is normally the application level of the host system, there may be problems performing the required services. The monitoring process must be able to start and stop processes for FDPS users. This may require more access than application programs normally have. Also, the monitoring process must be able to access services for a remote request. This may require exten-

ded access or the ability to start up a process for that user on the host system.

A problem faced by a DOS in a heterogeneous system is that of providing a uniform interface for FDPS users. With the base-level approach, this problem is lessened by the fact that the local operating systems are designed to support distributed systems. The local services provided by the various machines in the system should appear similar to the DOS. Guest systems, however, must take the various environments presented by the host operating systems and transform them into the single environment presented by the FDPS. One utility that is required here is a common command language that can be used by the distributed user to interact with the system. Using this command language, the FDPS user should be able to work with the system in a uniform manner regardless of the location in the network of the services required [GRAY79]. Because of the nature of a FDPS, the command language required must support services such as creation of processes and the ability to specify interconnections of processes. If the existing command language of a host machine does not support these functions, then the implementation of a distributed command language may not be a simple translation from the FDPS command language to the host command language.

In addition, there may be other differences between the host systems, which the DOS must hide. This requires the DOS to be responsible for hiding differences in representation of data and inconsistencies in services provided by the various host systems in addition to the services normally provided by the DOS. The major problem is the naming of services and resources. Each of the local operating systems provides its own local name space or name spaces, each with its own conventions. The DOS makes services available to the user by logical names which reveal nothing of the service's location. Distributed processes must also be able to locate services and other processes without regard to the service's or process' location. This requires the DOS to provide a global name space. The global name space must be able to handle cases such as generically named services and replicated files.

One approach to this problem is to have the DOS provide a directory service ([FORS80] and [PEEB80]). This service will perform translations from FDPS names to local names. The host then would receive requests using these local names. This approach allows other information to be included with

names, such as access information.

Providing communication among distributed processes is another task which is complicated by heterogeneous host systems. This service must provide conversion of data between two hosts if necessary. This may include such simple functions as translating from one character code to another. It may also involve more complex issues. For example, in the Desperanto system, software is represented as modules, which in turn are composed of data objects and functions which operate on them. The data objects may be represented in different ways on the various hosts in the system, but the distributed process should not be aware of this. A solution presented in [MAMR82b] is to provide an intermediate representation for data object and have the DOS perform the conversion from the local representations to the intermediate representation and vice versa.

4.6.4 Proposed Research

Research into this capability will form that basis of one of the two proposed testbeds. We propose a project to construct a guest operating system to run on at least two machines. In conjunction with this project, the separate areas of data management and file management, interprocess communication, and command languages will be addressed. Once complete, the testbed will support research into these areas, as well as resource management and load management.

4.6.5 Relationship to Other FDPS Work

The design of a FDPS using the guest system approach may provide not only information as to the feasibility of this approach, but also provide insights into the implementation issues of the base-level approach. Consideration should be made as to the level of support of FDPS services which may be expected from existing operating systems. Indeed, the criteria which make one operating system more suitable as a host system than another should be explored. And since operating systems may provide more support in one class of services than another, identification of the relative importance of each class to the FDPS is important. Information in these areas may allow the selection of host systems which are more suitable to the FDPS [FORS80]. Also, clarification of these criteria may provide insight into the design of new host operating systems for support of FDPS.

4.6.5.1 Distributed Software Tools - DSWT

Work of specific interest to this topic is the "DSWT Project." DSWT consists of one or more software tools subsystems (SWT) which communicate to locate and utilize resources and make decisions. DSWT takes the "meta approach" to the design and implementation of a network operating system. DSWT will give us a distributed environment for the network of PR1MES in the ICS computer lab. The DSWT project will be extended to a heterogeneous environment where other nodes will have implemented the entire set of tools of perhaps only a subset.

4.6.5.2 Distributed Compiling Shells

The Shell in an operating system is the Command Interpreter, the component of the operating system which parses the user's command line, instantiates the appropriate processes, and sets up communication between them, monitors their execution, takes appropriate steps when errors occur, and "cleans-up" when the processes terminate. In most systems, shells are interpretive; that is, they parse one user command, instantiate the correct processes to carry it out, and when they have terminated, a return is made to the command interpreter in order to carry out the next user command. In a Fully Distributed System, this is inappropriate since it is intended to take advantage of the inherent parallelism of the system by executing user jobs as concurrent systems of processes, executing in parallel. Therefore a new style of shell must be developed which takes in an entire user's command file, consisting of several command lines, parsing the entire command file, from which a task graph as described above, can be built, and distributing the results of this parsing step, as subgraphs, to the Local Operating Systems which have to carry out each subgraph derived from the central task graph.

4.6.6 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (6 m-m/year)	12
Programmers (3 at 5 m-m/year)	30
Secretarial Support (3 m-m/year)	6

Equipment

Computer Time — Substantial on heterogeneous environment

Timing

First period of 18 months:
Research into capabilities required.
Construction.

Last period of 6 months:
Experimental use of testbed.
Evaluation.

4.6.7 References

[ENSL78] P. Enslow, "What Is A 'Distributed' Data Processing System?," ul Computer], Vol. 11, No. 1, pp. 13-21, Jan. 1978.

[FORS78] H. Forsdick, "Operating Systems for Computer Networks," ul Computer], Vol. 11, No. 1, pp. 48-57, Jan. 1978.

[FORS80] H. Forsdick, W. MacGregor, R. Schantz, and R. Thomas, "Distributed Operating System Design: Phase I," BBN Report No. 4455, Aug. 1980.

[GELL77] D. Geller, "The National Software Works -- Access to Distributed Files and Tools," ul Proceedings of the ACM Annual Conference], pp. 39-43, Oct. 1977.

[GRAY79] T. Gray, "Network Job Control: The Tower of Babel Revisited," Phd. Thesis, UCLA-ENG-7901, Jan. 1979.

[KIMB76] S. Kimbleton and R. Mandell, "A Perspective on Network Operating Systems," ul AFIPS National Computer Conference Proceedings], Vol. 45, pp.551-559, 1976.

[MAMR82a] S. Mamrak and D. Leinbaugh, "Desperanto: Software Support for Distributed Processing," submitted to ul Computer Networks], Feb. 1982.

[MAMR82b] S. Mamrak, J. Kuo, and D. Soni, "Supporting Existing Tools in Distributed Systems: The Conversion Problem," submitted to the 3rd International Conference on Distributed Computer Systems, Miami, Florida, Oct. 1982.

[MILL77] R. Millstein, "The NSW: A Distributed Processing System," ul Proc. of the ACM Annual Conference], 1977.

[PEEB80] R. Peebles and T. Dopirak, "ADAPT: A Guest System," ul Proceedings of the IEEE Spring Compcon], pp. 445-454, Feb. 1980.

[THOM78] R. Thomas, R. Schantz, and H. Forsdick, "Network Operating Systems," BBN Report 3796, Mar. 1978.

[WATS80] R. Watson and J. Fletcher, "An Architecture for Support of Network Operating System Services," ul Computer Networks], Vol. 4, No. 1, pp. 33-49, Feb. 1980.

4.7 THE NETWORK ARCHITECTURE --- STANDARD PROTOCOLS AND INTERFACES

4.7.1 Description

The "network architecture" is the master plan defining the rules governing the overall structure of the distributed system at all levels of interaction. The network architecture defines how resources will be provided and utilized in a distributed processing environment. The network architecture consists of the complete definition of the following items:

- Standard interlayer interfaces
  - Interface data units
  - Service data units
  - Interface control information
  - Services provided across interface
  - Service request formats (procedure calls)
- Standard peer protocols
  - Protocol data units
  - Protocol control information
  - Communication standards

4.7.2 Background

Experience has shown that the definition and enforcement of a completely defined network architecture is essential to the development of "good" distributed systems. Initial efforts in these areas have usually followed the path of *ad hoc* design, and the results clearly reflect this approach.

The need becomes especially apparent when there is a requirement to extend or expand the system.

4.7.3 Problems

There are a number of problems in this area that must be addressed.

- It is impossible to completely define the network architecture prior to system implementaton because of the inability to identify a priori of all of the system features that must be defined.
- It is difficult to verify compliance with either interface or protocol standards.
- It is difficult to verify completeness and accuracy of interface and protocol definitions.
- A major problem is educating the designers *and* implementers as to the pervasiveness of the network architecture definition. The *de facto* definition of the architecture by "unrelated" and "uncontrolled" "low-level" decisions must be prevented.

4.7.4 Proposed Solution

There are no good models or examples of solutions to this problem --- even poor ones.

The recommendation is that the basic feature of the network architecture be defined as early as possible and that a "network architecture administrator" be established to continually monitor the development of the network architecture definition.

4.7.5 Relationship to Other FDPS Work and SSC's

Work in this area should be started at the earliest point possible and continue in parallel with all stages of the development of a distributed system.

4.7.6 Resources and Schedule

To cover a 24 month period:

Manpower	man-months
Senior Staff (2 m-m/year)	4
Junior Staff (3 m-m/year)	6
Secretarial Support (4 m-m/year)	8

4.8 OPERATIONAL SUPPORT CONCLUSION

We have discussed six capabilities for the operational support of distributed systems. While these capabilities can be studied independently, we propose that two testbeds be created as vehicles for research. The design and construction processes will provide a basis for research into capabilities such as file systems and data management, interprocess communication, and command languages, while the completed testbeds can be used for experimentation into capabilities such as interprocess communication, command languages, resource allocation, and load management. Note that these research issues overlap--some capabilities will benefit from both the construction process and the completed systems. The construction process will also provide excellent opportunity for study of the process and techniques for constructing distributed operating systems.

The two testbeds differ fundamentally in their approaches. The first takes are 'guest' approach, basing the distributed operating system implementation on application programs run by existing host operating systems. This approach is essentially based on expediency, and suffers inherent limitations in its capabilities. Nonetheless, guest systems have promise, because they emphasize heterogeneity and can couple considerably different equipment. We propose that we implement a testbed simultaneously on a Prime 550 and a VAX 11/780. Funds for computer usage are included in the estimates.

The second testbed is to be a 'native', or resident distributed operating system. In the current economic climate, but wishing nonetheless to do the experiment with more than trivial machinery, we have allowed funds for five (5) Perq workstations from Three Rivers Computer Corporation, to serve as a base for construction. These machines were designed for this environment, and have many advantages which make them eminently suitable - principally, the ability to redefine the machine architecture through microcode.

Both testbeds, during and after the design and construction phases, will facilitate the study of most of the capabilities described in this report. This feature makes the testbed concept extremely profitable.

4.8.1 Existing Research At Georgia Tech

The testbed construction process corresponds to two projects currently underway at Georgia Tech. The first, the 'guest' system, corresponds to the Distributed Software Tools project, under Professor R. J. LeBlanc. The

second approach, the 'native' system, corresponds to the Clouds project, under Professor M.S. McKendry (See Appendices H,I,J). Currently, neither project receives targetted funding, a factor that constrains the rate of progress.

The Clouds project currently uses two Perq workstations. We request five (5) additional stations, to aid during development for implementation, and after development for experimentation.

4.8.2 'Guest' System Resources

To cover a 30 month period:

Manpower	man-months
Senior Staff (4 m-m/year)	10
Junior Staff (9 m-m/year)	22.5
Programmers (5 at 5 m-m/year)	62.5
Secretarial Support (6 m-m/year)	15

Equipment

Computer Time	*Substantial or heterogeneous* testbed

Timing

First period of 18 months:
Research into capabilities required.
Construction of testbed.

Last period of 18 months:
Experimental design and use of testbed for experimentation.

4.8.3 'Native' System Resources

To cover a 36 month period:

Manpower	man-months
Senior Staff (4 m-m/year)	12
Junior Staff (9 m-m/year)	27
Programmers (6 at 5 m-m/year)	90
Secretarial Support (6 m-m/year)	18

Equipment

Computer Time — Substantial on testbed of 5-8 work stations

Timing

First period of 24 months:
Design 'global' operating system;
study support capabilities.

Last period of 18 months:
Experimental design;
Execute experiments in 'dynamic' capabilities.

SECTION 5

SUMMARY

5.1 USER ROLE IN DEVELOPMENT OF SUPPORT CAPABILITIES

It is almost an unnecessary repetition of a truism to state that developers need solid and immediate user feedback on the functionality provided, the design, and the utilization of any support capability while it is being developed. However, achieving this goal is at best moderately difficult; more often it is almost impossible.

As Gary Nutt said in [Riddle] "The user interface to software development tools is sometimes as important as their integrated functionality." Our experience at Georgia Tech indicates that an even stronger statement is more appropriate. The user interface is usually the single most important factor governing the overall value of any software development tool or design facility.

If the support capability in question is a simple, self-contained unit with reasonably well-defined input and output such as a text editor, user feedback can be obtained as increments of the support capability are developed. In the example of the editor, additional features can be added, command syntax can be changed, and output/display formats can be changed incrementally with relative ease. Also, prototypes of the intermediate products can be released to users for actual use and evaluation to provide guidance in the development and refinement of later versions.

On the other hand, if the support capability is a large and complex facility such as a simulator or data base design analyzer, it is very difficult to obtain user feedback at "intermediate-stages" of development. There is usually nothing to utilize, even on a trial basis, until the support facility has been completely implemented. This point is extremely important and applicable to many of the support facilities covered in this report since users will have had little experience with similar tools on which to base intermediate judgements.

Comments similar to those given in the paragraph above also apply to obtaining user feedback on the operational support capabilities. Only in this case the problem is even worse for now the complete operational system must be implemented, at least in prototype form, to allow user evaluation.

Providing user input in the form of detailed performance and operational requirements specifications becomes increasingly important for the three major classes of support capabilities discussed here, i.e. software development tools, design support facilities, and operational support capabilities; but, at the same time it becomes increasingly difficult to define the specifics of that user input.

5.2 INTEGRATION OF SUPPORT CAPABILITIES

There are several groupings or collections of support capabilities that are closely related to one another either in function or with respect to input and output. The desirable goal for organizing these related "tools" and establishing their interrelationships has been referred to as "integration." However, "integration" has been used to describe at least three different levels or methods of organization (Tom Love in "Discussion" in [Riddle]):

- The tools reside on the same system (a "toolbox")
- The output of one tool is valid input to another (a "workbench")
- Each tool has knowledge of what other tools may have done or be capable of doing (a "capable assistant")

Our experience at Georgia Tech has shown that the features and capabilities of at least the second level are essential. It is extremely convenient for the output of one tool process to be directly acceptable as input to another. Two of the major obstacles to user acceptance of individual software development tools have been

- Peculiarities (i.e., non-compatible differences) in the formats of their input and output, and
- Peculiarities (i.e., non-uniformity) in the syntax and semantics of their command languages and other aspects of the user interfaces required to utilize to "tool".

As important design goal is to avoid both of these forms of "peculiarities." In addition, if all tools utilize a single, common format for both input and output, the usability of the various tools is greatly augmented by flexibility in the intereconnection of various tools.

In addition to the ability to freely interconnect software development tools by the transfering of "output products" to other "inputs", there should be a hierarchy of support capabilities that provide a transfer of information between the various tools and other capabilities. Achieving this is certainly going to be more difficult than providing simple inteconnectivity.

5.3 IMPORTANCE OF PRODUCTIVITY AS A GOAL

Perhaps more fundamental than any other aspect when considering the evaluation of system support capabilities is the cost vs. increased productivity tradeoff. Whether viewing single processor systems, multiprocessors, or fully distributed systems it is cost of the time, labor, etc. in creating a tool (whether analysis based, design or implementation based, or other) and its expected payoff (in programmer productivity, machine throughput, communication costs or other) which determine whether the support capability is worth creating.

Examining the history of both programming languages and operating systems shows that over time it has become desirable to raise the "intelligence" level of these support tools. In both cases, we have delegated more and more of the lower level details to the machine itself. For example, we first had machine code, then assemblers, then compilers, then compiler compilers, etc. This trend is presently moving faster than at any other time, for the capacities of modern machines are rising so rapidly. **We simply expect more from computers now.**

The usual support tools of most present-day commercial systems are very primitive compared to what is dictated by the cost vs. utility tradeoff discussed above. Some of this "poor" support could be changed rather easily, others do not as yet have enough productivity value to overcome their costs. For example, even though bit map displays (raster scan) and positioning devices are clearly faster to use, their current cost has prevented widespread use. Whether in hardware or software, it is important to locate the support items which will clearly maximize the overall gain.

Unfortunately, distributed systems design, implementation, and operation are still very much research topics. It appears to us that our ability to accurately predict productivity payoffs for support capabilities implementing new concepts in not yet defined environments is not yet feasible.

5.4 TRANSPORTABILITY OF SUPPORT CAPABILITIES

It is highly desirable to have the ability to be able to transport all of the distributed system support capabilities discussed here from one operating environment to another. The types of obstacles inhibiting this are not very much different from those encountered with centralized systems for the software development support tools; however, the problems are quite different as to both scope and magnitude with respect to the design support and operational support capabilities.

We at Georgia Tech have had a large amount of experience in constructing an integrated set of software development tools (the Georgia Tech Software Tools Subsystem) and "transporting" that subsystem from one environment to another. The problems of incompatible language implementation and features can be overcome fairly easily, especially utilizing the editors available in the tool set. The major problems encountered have to do with transporting the process control involved and defining a suitable standard format that can be utilized for both the output and input of each individual tool.

Some of the design support capabilities, such as simulators, can be transported with a reasonable amount of work. However, those capabilities that interact directly with the target system, such as monitors, are probably not transportable at all. Of course, the "concepts" are transportable. It is just the implementation that is probably too specialized for use elsewhere. Support capabilities such as the designer workbenches can probably be transported with a reasonable amount of effort to run on a different processor. However in this case, the characteristics of the target environment may be deeply embedded in the details of operation of the workbench processes.

Operational support capabilities are, by this very nature and purpose, highly oriented to a specific target environment. Again, it probably is only the concepts that are easily transferred.

5.5 EVALUATION OF SUPPORT CAPABILITIES

The evaluation of support capabilities for distributed systems is not much different from the evaluation of similar support for centralized systems. The target environments certainly have major differences, but evaluation of capabilities such as these is most often heavily influenced by the "user side" rather than the "output side."

Evaluation most often relies primarily on subjective ratings of factors such as

- Learnability
- Utility
- Functionality provided/supported
- Reliability of tool operation and product
- Performance of tool and its produce
- Integration of various tools
- User acceptability (for instance compared to "Not invented here" problems)

5.6 DEVELOPMENT OF OPERATIONAL SUPPORT CAPABILITIES

Research on distributed systems at Georgia Tech (as well as elsewere) has indicated that it is difficult to proceed past even the most rudimentary research without the experience of designing, building, and operating a distributed system. While we have been able to do some experimentation on our initial FDPS testbed consisting of five Prime computers, we have concluded that a hardware/software testbed designed exclusively for experimentation is required.

The Clouds project is undertaking the design and construction of this testbed. Clouds is being constructed for a group of Perq workstations connected by a 10Mbps Ethernet. Since the Ethernet also links other equipment in our computer laboratory, in particular the Prime computers, the new testbed will be fully integrated with existing facilities.

The Clouds project will proceed in three phases. During Phase 1, design and initial construction, operational support capabilities will be studied and developed. Once the initial testbed is functional, Phase 2 will entail evaluation and refinement of operational support capabilities, and development of software support capabilities. Finally, Phase 3 will involve the exploitation of the testbed. This will involve study of all support capabilities, and will also involve experimental research into real time control systems, personal computing environments, office automation systems, and distributed databases, which are all applications of the testbed.

A Clouds status report is included as Appendix H of this document.

5.7 ROLE OF NETWORK ARCHITECTURE

The requirement for a network architecture that fully defines standard protocols and interfaces was discussed in Section 4 as a specific operational support capability. The critical importance of the network architecture to the overall success of any distributed processing system indictes that it should be given much more attention than just consideration as "another" operational support capability.

The development of our ability to organize the specification of the design of a distributed system is probably the most important advance made in this area since the inception of the concept of distributed processing systems. At least one of the authors of this report has been involved with the design implementation and operation of a number of distributed processing systems, and he feels that the development of a good network architecture and its control through the development cycle is the most important contribution to overall project success.

5.8 DEVELOPMENT PRIORITY FOR SELECTED SUPPORT CAPABILITIES

There are a number of criteria that one might utilize in developing a ranking of importance of the various individual support capabilities described in this report. Some of these criteria are:

- Difficulty of research problems to be addressed and solved.
- Length of time required for complete development of the support capability.
- "Position" of the specific support capability on the "critical path" of the overall system development schedule.
- Anticipated "value" of the support capability in improving system performance consider factors such as
  - Response time
  - Reliability
  - Robustness/Fault-Tolerance
  - Resource utilization
  - Effectiveness of system control
  - etc.

The major problem facing any attempt to prioritorize the support capabilities is that almost none of the criteria listed above produce the same answers; in fact, some of the criteria are internally inconsistent in the ordering they suggest. Further, several of the criteria are directly contradictory.

5.8.1 Criteria Utilized in This Report

Since one of the original goals of this project was the actual implementation of the "highest priority" support capabilities, the criteria utilized here to order them has been their position on the critical path --- just how essential is the capability in an actual implementation. In the lists given below, those capabilities designated "highest priority" represent the minimum subset essential to implement a basic version of a loosely-coupled distributed processing system.

5.8.2 Priority List

- Highest Priority --- Essential/First
  - Standard Architecture, Protocols, and Interfaces
  - Distributed Systems Testbed
  - Distributed File and Data Management System
  - Language support for Robust Distributed Programs
  - Compilation Techniques for Distributed Programs

- •• Distributed Resource Management
  - ••• Distributed Resource Access
  - ••• Distributed Resource Allocation
  - ••• Distributed Process Execution
- •• Distributed Load Manager
  - ••• Distributed Command Language (initial capabilities only)
- •• Distributed Interprocess Communication
- •• Distributed Execution Monitor (IPC monitoring as a minimum)

• **Lower Priority --- "Highly Useful"**

- •• Distributed Command Language (Full capabilites)
- •• Distributed Design Language
- •• All remaining System Design Support Facilities

• **Lowest Priority --- "Also Useful"**

- •• Distributed Compilers
- •• Compiler Development Tools for Heterogeneous Systems
- •• Software Version Management
- •• Cost Estimation and Control
- •• Guest System Testbed

5.9 REFERENCES

[Riddle80] Riddle, W.E. and R.E. Fairley, Software Development Tools, Springer-Verlag, Berlin, 1980, 277+viii pp. (Proceedings of a workshop on Software Development tools held at Pinegree Park, Colorado, May, 1979. Workshop emphasized pre-implementation phases of Software development.) Of particular interest:

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

FILMED